Efficient Notification Ordering for Geo-Distributed Pub/Sub Systems

Roberto Baldoni, Silvia Bonomi, Marco Platania, and Leonardo Querzoni

Abstract—A distributed event notification service (ENS) is at the core of modern messaging infrastructures providing applications with scalable and robust publish/subscribe communication primitives. Such ENSs can route events toward subscribers using multiple paths with different lengths and latencies. As a consequence, subscribers can receive events out of order. In this paper, we propose a novel solution for ordered notifications on top of an existing distributed topic-based ENS. Our solutions guarantees that each pair of events published in the system will be notified in the same order to all their target subscribers independently from the topics they are published in. It endows a distributed timestamping mechanism based on a multistage sequencer that produces timestamps whose size is dynamically adjusted to accommodate changing subscriptions in the system.
An extensive experimental evaluation based on a prototype implementation shows that the timestamping mechanism is able to scale from several points of view (i.e., number of publisher and subscribers, event rate). Furthermore, it shows how the deployment flexibility of our solution makes it perform better in terms of timestamp size and timestamp generation latency when the system load exhibits geographic topic popularity, that is, matching subscriptions and publications are geographically clustered. This makes our solution particularly well suited to be deployed in geo-distributed infrastructures.

Index Terms—Total order, Publish/Subscribe, Geo-Distributed Systems, Logical timestamps, Event based communications, Geographic Topic Popularity.

1 Introduction

Modern large-scale services are usually built on top of asynchronous communication primitives able to mask the unreliability of low-level networks and the dynamism of the application participants by decoupling the interacting parties in space and time. The publish/subscribe paradigm provides communication services where message addressing is implicitly handled by an *Event Notification Service* (ENS), a middleware infrastructure that matches the content of events produced by publishers against interests expressed by subscribers in the form of subscriptions.

Many research efforts in publish/subscribe systems focused on reliability and performance aspects with few contributions in the area of event ordering [1]–[5]. Defining a coherent specification for notification ordering is a fundamental step for a wide range of applications like stock tickers, messaging, command-and-control, or those based on composite event detection [6], where specific event patterns must be concurrently detected by distributed and possibly independent application components. As an example, in distributed online games [7] users in close proximity in a virtual world are supposed to see events happen in a single consistent order. In Electronic stock tickers [8], investors are required to see coherently ordered stock price evolutions to take their investment decisions.

In this paper we consider the following ordering problem: how to guarantee that two subscribers sharing subscriptions with common interests are notified about events matching those subscriptions in the same order. While the above ordering problem stems from the simple rationale that two participants should always see the notification of two events in the same order, its enforcement in distributed ENSs is far from being trivial. Violations to the ordering property can easily arise due to the fact that two events, possibly published by different publishers, can follow distinct paths through the ENS before reaching the points where they will be notified to the final recipients. Furthermore, non-determinism, in the form of unpredictable network latencies and message losses, can easily exacerbate the problem, especially in geo-distributed application scenarios.

Current solutions either require complex offline setups that must be continuously updated when subscribers change their interests [2], or give up some ordering aspects only guaranteeing per-source ordering [9], or are based on synchronization among processes in order to deterministically order conflicting events (i.e., [1], [4], [5]). Such synchronization approaches, either hardware-based [1], or based on total order [4] among all processes receiving an event (using centralized sequencers or distributed consensus primitives like Paxos [10]) or, finally, characterized by the widespread usage of explicit acknowledgments [5], end up into scalability problems with respect to performance when both the

• *R. Baldoni, S. Bonomi, and L. Querzoni are with the Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Rome, Italy.*
M. Platania is with the Department of Computer Science, Johns Hopkins University, Baltimore MD, USA
E-mail: {baldoni|bonomi|querzoni}@dis.uniroma1.it; platania@cs.jhu.edu

A preliminary and short version of this paper has appeared in the proceedings of the 26th International Parallel & Distributed Processing Symposium.

event rate and the number of entities involved in the synchronization increase [11] or as soon as WAN links are involved [12].

This paper focuses on studying event ordering for geo-distributed publish/subscribe communication middleware looking for a scalable solution in terms of number of published events and subscriptions, while accommodating subscription changes at run-time. More specifically, the paper introduces a novel timestamping solution designed to be used with topic-based publish/subscribe systems. The purpose of our solution is to generate logical timestamps that can be used, on receiver side, to enforce the following *total notification order* (TNO) property without any explicit synchronization: if two independent subscribers are notified about the same two events, then these two events will be notified to them in the same order[1].

The core of our solution is a new logical timestamping mechanism based on subscribers' interests. The structure and size of each timestamp is automatically calculated at run-time on the basis of current subscription overlapping, in order to keep it to a minimum (as it can vary between a single integer and a vector of integers whose size is the number of topics). Timestamps are built through a multistage sequencer based on a distributed architecture. A key feature of this architecture lies in its deployment flexibility: multiple stages of the sequencer can be colocated on machines deployed in different sites of a geo-distributed infrastructure (e.g., a service provider with multiple data centers). This flexibility allows system designers to exploit specific locality characteristics enjoyed by many large-scale geo-distributed applications, such as YouTube[2], to drastically reduce both timestamp size and generation latency. Geographic topic locality can indeed make most of the overlapping among subscriptions local to a specific geographic area increasing the probability that a timestamp will be entirely generated within a single site of the distributed timestamping architecture, thus avoiding costly inter-site (i.e., WAN) communication.

From an architectural point of view, the timestamping mechanism, encapsulated within a software component that can be deployed on top of existing reliable topic-based publish/subscribe middleware, transparently delivers events notified by the ENS to the application layer guaranteeing the total notification order. When deployed on top of a non-reliable ENS, our mechanism is able to deterministically tag every event whose notification violates the TNO property; this gives application developers the possibility to treat out-of-order events in an appropriate manner.

The performance of our solution have been analyzed through an extensive experimental evaluation. The results show how it creates timestamps whose size scales with respect to the number of subscribed topics. We developed a prototype implementation of our solution, in order to study its behavior in a realistic geo-distributed setting, mimicking a common architecture employed by cloud-providers and by large companies with several data centers. The experimental evaluation of the prototype takes into account a *geographic topic popularity*, in which subscriptions and publications are geographically strongly clustered [13], and a *spray-and-diffuse* pattern, in which interest clustering is mildly present [13], [14]. Results show that an increasing matching between topic popularity and locality of interest produces a timestamp generation latency of less than 300 ms in the presence of intense publication rates (up to 10000 events/sec).

The rest of this paper is organized as follows: Section 2 introduces the system model and states the problem explaining why its solution includes several difficult aspects; Section 3 describes our algorithm; Section 4 presents some important engineering aspects; Section 5 reports the results of the evaluation of our solution; Section 6 explains how the problem of ordering events has been tackled in the literature and, finally, Section 7 concludes the paper.

2 SYSTEM MODEL AND PROBLEM STATEMENT

We consider a system composed by a number of interacting clients that can act as publishers (data producers) or subscribers (data consumers). Clients exchange data in the form of events using a topic-based selection model, thus we assume that they share a common knowledge on a fixed set of available topics. Each piece of data produced by a publisher is published on one of the available topics and takes the form of an event. Each subscriber issues a subscription S containing the set of topics it is interested in. An event e published on a topic T matches a subscription S if and only if $T \in S$; when this happens, the corresponding subscriber must be notified about e. Clients do not interact directly: their interactions are mediated by an *Event Notification Service* (ENS) that exposes the fundamental interface of a publish/subscribe system, i.e., the *publish*, *subscribe/unsubscribe* and *notify* primitives. Without loss of generality, here we assume that the ENS is implemented as a distributed middleware.

In addition, in order to simplify the description of our solution, we will initially assume that our system works on top of a reliable communication substrate, that all communication links deliver messages in FIFO order, and that all processes are correct. Section 4 details how some of these assumptions can be removed or relaxed.

The ordering property we want to enforce is defined as follows:

Property 1: TOTAL NOTIFICATION ORDER (TNO). Let e_i and e_j be two distinct events notified to a subscriber s. If e_i is notified to s before e_j, no subscriber will be notified about e_i after being notified about e_j.

1. The TNO property, also known as *Pairwise Total Order*, is considered in the literature as one of the strongest form of ordering achievable in distributed publish/subscribe middleware [5].

2. Brodersen et al. [13] showed that at least 40% of YouTube videos have 80% of their total views coming from a single country, indicating strong user interest locality.

Note that this definition matches the definition of *Weak Total Order* given in [15] in the context of total order specifications [16]. Differently from those specifications, we do not consider any form of deterministic agreement (uniform or not uniform) because here we are only interested in designing an ordering layer to be transparently plugged on top of a generic ENS which can provide different reliability and agreement properties. The TNO property also matches the *pairwise total order* property defined in [5].

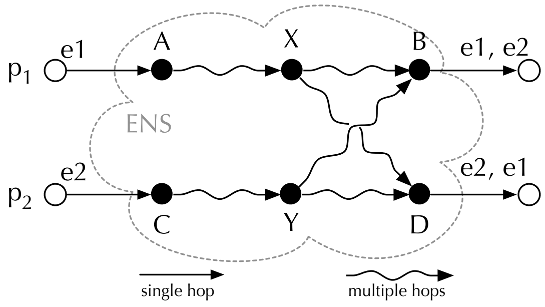

Fig. 1. An example showing how notifications can be performed out of order in a distributed event notification service.

Guaranteeing TNO in a distributed setting is a complex task. As an example, consider the toy system depicted in Figure 1: the six black dots represent processes constituting the ENS, the white dots on the left (p_1 and p_2) are two publishers and those on the right (s_1 and s_2) are two subscribers. A common solution for ordering events published on a specific topic is based on the usage of sequencers: a single node in the ENS is elected as a "sequencer" for all the events published in that topic. In our example X acts as the sequencer node for topic $T1$, receiving all the events published in $T1$ (i.e., event e_1 published by p_1), adding a sequence number to them, and then routing the events toward the intended destinations (i.e., s_1 and s_2 notified by nodes B and D). Similarly, Y is the node in charge of sequencing events published in topic $T2$ (event e_2 in the example).

This simple approach, however, is not useful when subscriptions intersect in multiple topics. For example, assume that both s_1 and s_2 are subscribed to $T1$ and $T2$. In this case the sequence numbers attached by X and Y would be completely uncorrelated and useless to check for a correct notification order on the subscribers' side. Centralized solutions, i.e., using a single sequencer for all the topics, have important scalability drawbacks. Similarly, distributed consensus algorithms impose stringent latency requirements to provide synchronization in a timely manner [11], [12]. Therefore, they cannot be realistically considered in scenarios where large scale or large loads are expected.

3 The Event Ordering Algorithm

In this Section we first introduce an abstraction to illustrate how the event ordering problem in publish/subscribe systems can be theoretically addressed, and the design principles underlying a possible distributed implementation. Then, we detail the algorithm that implements the proposed solution. The algorithm pseudocode and the related correctness proofs are reported in the appendix.

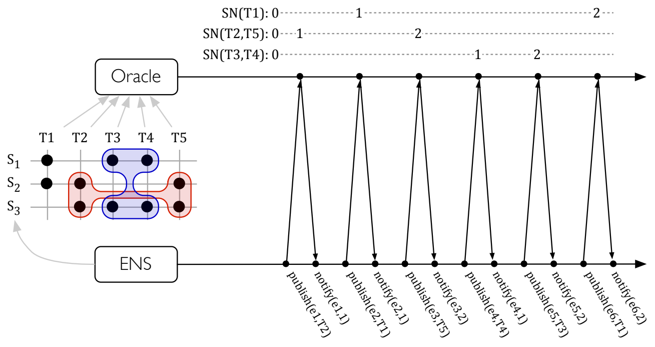

Fig. 2. The oracle assigns sequence numbers to events published in different topics on the basis of intersections among subscriptions.

3.1 Event Ordering Abstraction

From a theoretical standpoint we need an event ordering abstraction (*oracle* in the following) able to produce sequence numbers. Such oracle would provide, for each topic, the sequence number representing the timestamp of the next event published in that topic. Sequence numbers would need to be generated such that two events published on topics that are both subscribed by at least two subscribers have different comparable timestamps. To enforce this condition, and thus guarantee TNO, the oracle must check all subscription intersections and thus requires complete knowledge of subscriptions. Figure 2 shows an example where the oracle accesses knowledge of subscriptions from the ENS to identify intersections among s_2 and s_3 on topics $T2$ and $T5$, and among s_1 and s_3 on topics $T3$ and $T4$. As a consequence the oracle will maintain a sequence number for $(T2, T5)$ and one for $(T3, T4)$. A third sequence number will be maintained to timestamp events published in $T1$ only as events for this topic must not be ordered with respect to events published in any other topic. Thanks to these sequence numbers, events $e1$ and $e3$, published in $T2$ and $T5$ respectively, will be notified by all subscribers (s_2 and s_3) in the order defined by their timestamps (1 for $e1$ and 2 for $e3$), independently from possible reordering happening in the ENS during the event diffusion phase. Event e_2 published in topic $T1$ can be notified from s_2 independently from the former events (i.e., without a precise order), as the only other subscriber in $T1$, i.e., s_1, will not be notified about $e1$, nor $e3$.

In the following sections we introduce a distributed implementation of this oracle and show how to use it to guarantee TNO on the subscriber side. However, before delving in these design details, it is useful to clearly outline the design principles underlying our solution.

3.2 Design principles

A system implementing the event ordering abstraction should match the following design principles:

Full decoupling - It must retain the typical full decoupling characteristics of publish/subscribe, i.e., no direct interactions should happen between publishers and subscribers.

Support for large subscription loads - It must gracefully scale in scenarios with large number of subscriptions and multiple possible interest intersections. This must be achieved by both adopting short timestamps to reduce the overhead on the ENS and a clever internal design. The system should be designed in a distributed fashion with the aim of sharing the load imposed by timestamp generation on multiple machines (i.e., no machine should maintain full knowledge of the system state) and thus avoid possible bottlenecks.

Asynchronous one-way message flows - It must avoid any kind of explicit synchronization among its internal processes by adopting a one-way message flow strategy (i.e., no ACKs are required) for the most common operations, like event timestamping or subscription management; the lack of explicit synchronization is a crucial principle needed to support intense event publication rates [11].

Full independence from the ENS - It must be designed to be transparently pluggable on top of existing topic-based publish/subscribe middleware platforms. This principle facilitates the deployments within existing infrastructures.

Flexible deployment - It must allow the deployment of its internal components in a flexible manner so to efficiently support different application scenarios, from simple centralized setups to distributed deployments with geographic topic popularity, while maximizing the available resource usage.

3.3 Architectural aspects

Our solution assumes that all participants to the system (publishers and subscribers) are equipped with an *Ordering module* that implements the algorithm described in the next Section (see Figure 3). This module mediates the interactions between application level software components, that act as information producers (publisher applications) or consumers (subscriber applications), and a standard ENS.

We assume that the ENS implements a standard topic-based publish/subscribe interface (here represented by the *ENSpublish*, *ENSsubscribe/ENSunsubscribe* and *ENSnotify* operations). The same interface is offered by the ordering module to the application level, therefore

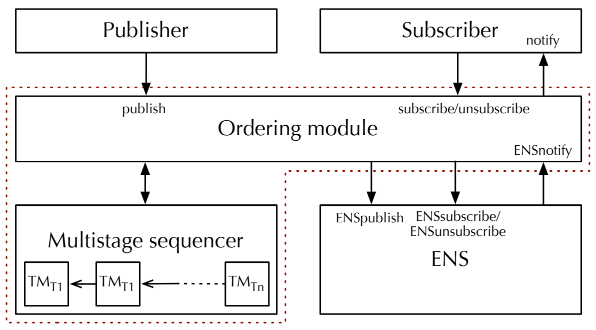

Fig. 3. Architectural view that shows how the Ordering module acts as a mediating software layer between the applications and an existing event notification service. The dashed line represents the whole timestamping mechanism that also includes a multistage sequencer used to produce timestamps.

neither the applications, nor the ENS must be changed in order to work with our solution. In the following of this section we will consider a deployment where our solution is coupled with a reliable ENS. In particular, we assume that when the ENS returns from an invocation to *ENSsubscribe*, all events published on the subscribed topic after that point in time will be eventually *ENSnotified* to the subscriber. Section 4 will provide further details on how the solution behaviour changes when it is coupled with a non-reliable ENS. Note that, thanks to the full adherence of the ordering module with the standard publish/subscribe interface, it could also be used in conjunction with a mapping layer that efficiently allows the adoption of a content-based interface on top of a topic-based publish/subscribe system [17].

The ordering module also needs to access a point-to-point communication primitive that can be offered by the operating system or by other solutions like an overlay network. We also assume that the set of available topics is fixed and a precedence relationship $\rightarrow$ holds among topic identifiers inducing a total order on them: if $T \rightarrow T'$ we say that T has a higher rank in the relationship than T'.

Ordering modules communicate with a multistage sequencer, constituted by a set of processes, one per topic, called *topic managers* (TMs). Finally, we assume there is a method to univocally map a topic T to its topic manager TM_T. This problem can be solved in several different ways, i.e., through a static mapping provided as a configuration parameter, using a DNS, or resorting to a distributed hash table as in rendez-vous based publish/subscribe systems [18]. In the following, whenever there is no ambiguity, we will use the terms *publisher* and *subscriber* to refer the parts of our ordering module located respectively at the publisher and at the subscriber.

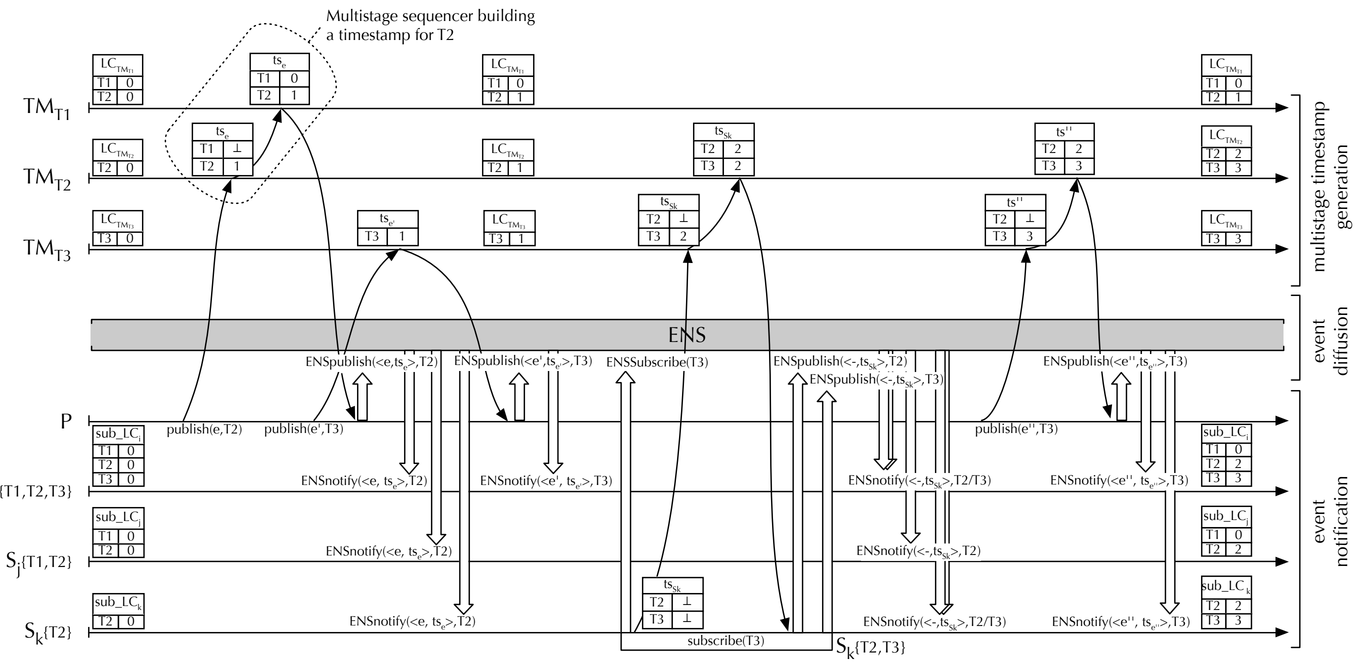

Fig. 4. Example of a system run with three subscribers, S_i, S_j and S_k, and a publisher P.

3.4 Algorithm description

The basic idea behind the algorithm is to assign a logical timestamp ts to each event. By looking at a timestamp, a subscriber must be able to decide if the event can be immediately notified to the application level or there is another event that precedes it in the TNO order. In this latter case the current event is locally buffered while the subscriber waits for the missing event to be notified by the ENS.

The algorithm to be executed when an event is published is split in three phases (Figure 4): (i) *multistage timestamp generation*, where a timestamp is generated for the event; (ii) *event diffusion*, where the ENS delivers the event and its timestamp to all the intended subscribers; and (iii) *event notification*, where subscribers, by looking at the timestamp content, decide if the event is the next in the TNO order to be notified. The algorithm uses only local information maintained by each process. A topic manager TM_T stores all subscriptions containing topic T, a sequence number LC_T that counts the number of events published in T and a (possibly empty) set of sequence numbers LLC_T storing identifiers and sequence numbers of topics $T', T'', \cdots$ with lower ranks than T. Each subscriber stores its subscription S and a set sub_LC containing the sequence number of the last event notified on T, for each topic $T \in S$ (i.e., it maintains a local subscription clock).

In the following, before describing the multistage timestamp generation procedure, we first provide a few formal definitions we will use later (i.e. *sequencing group*, *timestamp*, and *order relation between two timestamps*).

Informally, a *sequencing group* for a topic T contains the (ordered) set of all the other topics whose events must be ordered with respect to events published in T to guarantee TNO.

Definition 1: A *sequencing group* of a topic T ($\mathcal{SG}_T$) is a set of topics including T and all T' such that there are at least two subscriptions including both T and T'.

Therefore, $\mathcal{SG}_T$ is a one-way sequence of topics, whose direction is determined by the precedence relationship $\rightarrow$, and whose content may only change with subscription updates. A topic manager TM_T can calculate $\mathcal{SG}_T$ by looking at the list of subscriptions containing T it holds. Specifically, TM_T (i) calculates the union of all topics in the subscriptions it knows and (ii) removes all topics that appear in a single subscription (except T). The resulting set is $\mathcal{SG}_{\mathcal{T}}$.

Given the sequencing group of a topic T we can now define how a timestamp for events published in T must be structured. Informally, the timestamp contains a set of entries, one for each topic contained in the sequencing group of T. Each entry represent a sequence number for a specific topic $T' \in \mathcal{SG}_{\mathcal{T}}$.

Definition 2: Let e be an event published in a topic T, $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ the topic ordering according to the precedence relation $\rightarrow$, and $\mathcal{SG}_T$ the sequencing group of T. A timestamp ts_e for e is a set of pairs $< T_i, sn_i >$ ordered according to $\rightarrow$, where $T_i \in \mathcal{SG}_T$ is a topic identifier and sn_i is the sequence number for T_i.

Now that we know how a timestamp is internally structured, we can define when and how two timestamps can be compared.

Definition 3: Let ts_e and $ts_{e'}$ be two timestamps associated with two different events e and e'. We say that ts_e and $ts_{e'}$ are *comparable* if there exists at least one topic identifier present both in ts_e and $ts_{e'}$ (i.e., $\exists\ t_{id}, i, j \mid (< t_{id}, i > \in ts_e) \wedge (< t_{id}, j > \in ts_{e'})$).

From the three definitions above, it is easy to see that given two events e and e' published respectively in topics T and T', the corresponding timestamps ts_e and $ts_{e'}$ are comparable if and only if $\mathcal{SG}_T \cap \mathcal{SG}_{T'} \neq \emptyset$. The fact that their sequencing groups intersect means that a total order must be defined for events published within them.

Definition 4: Let ts_e and $ts_{e'}$ be two timestamps associated with two different events e and e'. We say that ts_e *is smaller* than $ts_{e'}$ (i.e., $ts_e < ts_{e'}$) if

1) ts_e and $ts_{e'}$ are comparable, and
2) $\forall < t_{id}, sn > \in ts_e \mid \exists < t_{id}, sn' > \in ts_{e'}$, $sn \leq sn'$, and
3) $\exists < t_{id}, sn > \in ts_e \mid \exists < t_{id}, sn' > \in ts_{e'}$, $sn < sn'$

As an example, in Figure 4 we show the timestamps for two published events e and e'. Considering the timestamp ts associated with e and the timestamp ts' associated to e' we have that they are not comparable.

Multistage timestamp generation: When the publication of an event e in a topic T occurs, the Ordering module on the publisher side contacts the topic manager TM_T to obtain a timestamp for e. TM_T starts a collaborative multistage timestamp generation procedure that involves the subset of TMs associated with topics belonging to the *sequencing group* of T. Specifically, TM_T:

1) Creates a timestamp structure with an entry for each topic T' such that $T' \in \mathcal{SG}_T$;
2) Increases its local sequence number LC_T;
3) Inserts this value in T's entry in the timestamp;
4) Inserts in the timestamp the values related to all the topics $\overline{T} \in \mathcal{SG}_T$ such that $T \rightarrow \overline{T}$;
5) Forwards the timestamp to the TM associated to the first topic in $\mathcal{SG}_T$ that precedes T according to the precedence relation $\rightarrow$.

Note that, given a specific order $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ among topics, the multistage timestamp generation flow proceeds in the opposite direction (i.e., given a topic T_i, TM_{T_i} will fill in the timestamp and forward it to some TM_{T_j} such that $T_j \rightarrow T_i$). In addition, based on the definition of *sequencing group*, the multistage timestamp generation is a one-way flow of messages, so to avoid loops among TMs that may prevent a correct timestamp construction [2] or create instability in large scale high-throughput systems [19]. The receiving TM also inserts in the timestamp the sequence number of the topic it manages, without increasing it. Then it forwards the timestamp to the next TM in $\mathcal{SG}_T$. When the last TM completes the timestamp, it is returned to the publisher that will publish the event in the ENS together with the timestamp. During this process each $TM_{T'}$ receiving a partially filled timestamp, uses its content to update the local clocks $LC_{\overline{T}}$ for all topics $\overline{T} \in \mathcal{SG}_T$ such that $T' \rightarrow \overline{T}$. Note that, in order to increase the scalability of the multistage sequencer, only event ids travel within requests, while event payloads are buffered on the publisher side while it waits for the multistage timestamp generation procedure to complete.

Figure 4 shows a run of the algorithm in a system with three subscriptions $S_i : \{T_1, T_2, T_3\}$, $S_j : \{T_1, T_2\}$, and $S_k : \{T_2\}$ and the precedence relation as $T_1 \rightarrow T_2 \rightarrow T_3$. The intersection of S_i and S_j, and the precedence relation determine the following *sequencing groups*: $\mathcal{SG}_{T1} = \mathcal{SG}_{T2} = \{T1, T2\}$, $\mathcal{SG}_{T3} = \{T3\}$. The publisher P publishes an event e on topic $T2$ and asks TM_{T2} to create the timestamp. TM_{T2} creates the structure of the timestamp with entries for topics $T2$ and $T1$, puts its sequence number in the timestamp and forwards it to TM_{T1} that, in turn, will complete the timestamp and return it to P. Finally P publishes both e and its timestamp on the ENS. Local clocks $LC_{TM_{T1}}$ and $LC_{TM_{T1}}$ are updated accordingly.

In the event notification phase, once an event e and its timestamp are notified by the ENS, the subscriber checks if the timestamp attached to the event is coherent with the event order maintained through the local subscription clock sub_LC. This check is performed by looking at the sequence numbers included in the timestamp: if the values for all the topics are equal to the corresponding ones stored locally in sub_LC_i, except for the topic where the event has been published, that must have a value greater than the local one by one unit, then no event with a smaller timestamp exists that must be still received by the subscriber, and the received event can thus be notified. Conversely, if a gap in the timestamp sequence is detected e is locally buffered. Buffered events are delivered as soon as all the events preceding them in the TNO order have been delivered. The assumption that the underlying ENS is reliable guarantees that buffered events will be eventually notified to the application level. Figure 5 shows an example where network lags induce a mis-ordering between two events $e1$ and $e2$ at S_k. A gap in the correct notification sequence is locally detected at the subscriber that buffers $e2$ waiting for the notification of $e1$ from the ENS. As soon as $e1$ is notified by the ENS, both events, in the correct order, are notified at the application level.

Timestamp properties: At a first glance, timestamps provided by the multistage sequencer resemble vector clocks, but they have very different structures. Vector clocks have a well defined and fixed structure that depends on the number of processes in the computation. On the contrary, our timestamp structure is related to topics rather than processes and its size depends on the current set of subscriptions. In particular, the size of a timestamp associated with events published in a topic T is equal to the dimension of $\mathcal{SG}_T$. In the best case $\forall T$, $|\mathcal{SG}_T| = 1$. This is the case, for example, in which

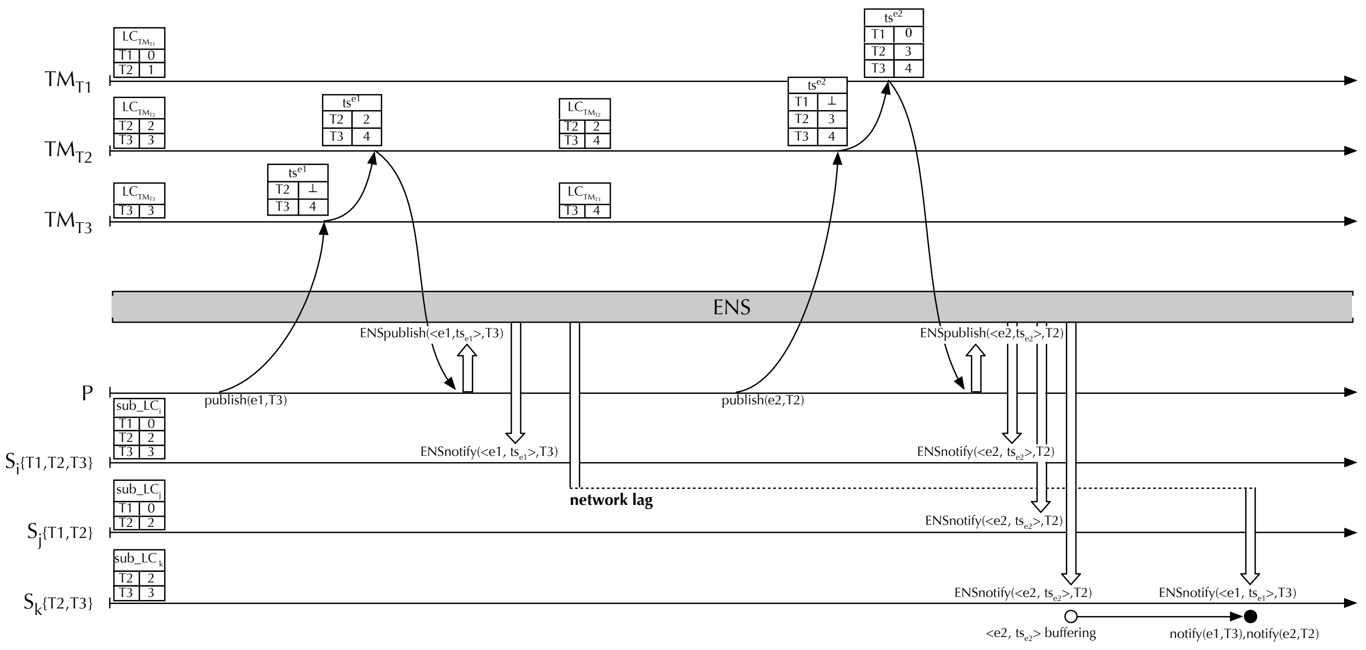

Fig. 5. Example of reordering for out of order notifications.

subscribers only subscribe a single topic each, thus avoiding intersections among subscriptions. In the worst case, instead, the size of $\mathcal{SG}_T$ equals the number of topics in the system, and this happens when all subscribers subscribe all topics. However, typically subscribers are interested just in a subset of topics. Hence, the timestamps size is expected to be much smaller than the topic cardinality. Typical applications are characterized by a total number of topics that is much lower than the number of participating processes. To this respect, our timestamps provide a more scalable solution to order events.

In addition, differently from vector clocks, where the ordering relationship holding among two events can be detected just comparing (entry by entry) the two vector clocks associated with the two events, with our timestamps the ordering relation can be detected looking firstly at the timestamp structure (i.e., the events have to be comparable according to Definition 3), and secondly, by examining the values contained in common entries of the timestamps. Let us finally remark that in our timestamping technique, the timestamp associated with an event does not bring any information about the producer of the event, thus preserving the space decoupling principle of the publish/subscribe paradigm.

Subscription management: Every new subscription (unsubscription) to (from) topic T induces a modification of $\mathcal{SG}_T$ and, then, of the set of TMs used in the multistage timestamp generation phase. Therefore, when a topic T is added to a subscription S, each topic manager $TM_{T'}$ associated with a topic T' in S must be advertised in order to let it recalculate $\mathcal{SG}_{T'}$ and thus avoid possible TNO violations. Furthermore, as soon as a subscriber subscribes a new topic, he will start being notified about events published in that topic. Due to the lack of synchronization between the time an event is timestamped and published and the time it is notified by the ENS to the intended recipients, the subscriber could both receive events timestamped before or after its subscription. Such events could possibly bring different timestamps (because sequencing have changed with its subscription). Those produced after its subscription are always comparable with other events published in different topics the subscriber is notified about. However, timestamps of events generated before its subscription could be non-comparable, making it impossible for the subscriber to infer the correct notification order. It is thus necessary for the subscriber to gather enough information before completing the subscription to deterministically discern events produced before it from those produced after.

To this aim, the subscriber first subscribes to T and, as soon as the invocation returns from the ENS, it starts buffering incoming events notified for T. Buffering at this stage is fundamental because only a subset of the buffered events, those published after the subscription, shall be notified in the right order at the application level. The subscriber then creates an empty subscription timestamp, containing one entry for each topic in the subscription. The timestamp and the new subscription including T are forwarded to the TM associated with the lowest ranked topic among the subscribed ones. Similarly to event timestamps, each entry of the subscription timestamp is filled in by the corresponding topic manager, which, however, also increases its local sequence number. In addition, each TM that receives a request, updates the list of subscriptions it holds. When the timestamp is complete, it is sent back to the subscriber, which uses it to update values contained in its local clock. Thanks to this subscription timestamp the subscriber is now able to clearly distinguish events produced before its subscription from those produced later, and can thus analyze the content of the buffer to discard old events and notify received events for T with timestamps greater than the one associated to its subscription.

Due to the fact that the subscription procedure increases the value of timestamps for all topics included in S and possibly changes multiple sequencing groups beside $\mathcal{SG}_T$, subscribers of these topics must be informed of this increase, otherwise they will start to endlessly buffer new event notifications as they suppose that the gap in the sequence number sequence is due to a missing notification. In order to avoid this undesirable side effect, the subscriber, after subscribing to the new topic also publishes a "dummy" event in all topics included in S attaching to it the subscription timestamp it just received. This dummy event has no application meaning, and thus it will never be notified to the application level, but its accompanying timestamp is used by all notified subscribers to correctly update their local clocks.

Figure 4 shows an example where subscriber k starts with subscription $S_k = \{T2\}$ and updates it at run-time by subscribing $T3$. Event e' published in topic $T3$ before k's subscription is timestamped with a single serial number added by TM_{T3} as $\mathcal{SG}_{T3} = \{T3\}$. When k starts its subscription, it first subscribes to $T3$ with the ENS ($ENSsubscribe(T3)$) and then sends a request to TM_{T3} containing a subscriptions timestamp with an empty entry for every subscribed topic, including also an entry for $T3$ (i.e., $\{T2, T3\}$ in the example). By receiving this request, TM_{T3} updates its local list of subscriptions, increases LC_{T3} and fills the corresponding entry in the timestamp with its value before forwarding it to TM_{T2}, which, in turn, will repeat the same operation and finally return the timestamp to k. By receiving the completed timestamp, k will use its content to update its local clock (that from now on will also contain an entry for $T3$) and will then flush the buffer selecting received events that must be notified at the application level (not shown in the example).

Finally, k publishes two dummy events in $T2$ and $T3$ that are notified to the corresponding subscribers (i, j, k and i, k respectively). The subscribers update their local clocks with the information contained in the event timestamp. The figure also reports the example of a third event e'' published in topic $T3$ showing how the topic timestamp has been dynamically adapted by the system to take into account new ordering opportunities with events produced in $T2$.

A similar approach is used to unsubscribe a topic T. However, during an unsubscription operation a subscriber can simply start to ignore further events received for T since the moment $unsubscribe(T)$ is invoked, thus avoiding any possible TNO violation. In order to maintain the system efficient, the subscriber must also inform all the relevant TMs about its subscription change. Without this step, in fact, sequencing groups would remain unchanged despite the possible removal of subscription intersections, and this would cause timestamps to be possibly larger than what is required by our solution to guarantee TNO. To this aim the subscriber sends to the TM associated to each of its subscribed topics a message containing its updated subscription, and to TM_T an empty subscription (because it is no longer subscribed to T, TM_T does no longer need to maintain its subscription). Upon receiving such messages TMs simply updates their local subscription lists, without updating any local clock.

4 Reliability and Ordering aspects

Working with unreliable ENSs - The algorithm introduced in Section 3.4 assumes that the ordering module is deployed in conjunction with a reliable ENS. While this is a desirable setup, sometimes adopting a non-reliable publish/subscribe middleware, i.e. a middleware that does not guarantee the notification of all events to all the intended recipients, may be preferable. As an example, non-reliable publish/subscribe middleware often sport better performance with lower end-to-end notification latencies, and are able to scale to larger sizes. This setup creates a simple but fundamental problem to our solution: whenever a gap in the sequence of events is spotted by a subscriber during the notification phase, the subscriber cannot decide if the missing event has been lost (because the ENS is unreliable) or its notification is just late. The strategy our solution adopts with respect to this issue is optimistic: every event notified in order by the ENS is notified to the application level, while late notifications are tagged as *out-of-order* and immediately notified to the application level. This leaves to the application developer the choice to discard these events or treat them in an appropriate way.

The main source of out-of-order notifications lies in the fact that two events, possibly published by different publishers, can follow distinct paths through the ENS, before reaching the point where they will be notified to the final recipients. To reduce the number of out-of-order notifications we can use again a buffering strategy on the subscriber side, but with some important differences in its management with respect to the solution shown in Section 3.4.

Every time the ENSnotify() primitive returns a new event e, the algorithm checks through the attached timestamp whether some other event may exist with a smaller timestamp. If there is a possibility that an event with a smaller timestamp exists but has not been delivered to the subscriber so far, then the event e is enqueued to a buffer able to host a maximum of b events and a timer for e is started (TTL_e). The event e is delivered through the notify() primitive when one of the following conditions holds: (i) all the events with smaller timestamps have been notified, (ii) TTL_e expires or (iii) the buffer is full, a new event must be buffered and e is at the head of the queue. By carefully tuning the buffer size and the timer length, system integrators can compensate for possible network delay fluctuations by paying some further end-to-end notification latency induced by event buffering.

Reliability - Making the algorithm presented so far work reliably in an environment where messages can be lost requires some minor changes. The loss of a message during the multistage timestamp generation phase, for instance, could lead a publisher to wait forever before publishing an event in the ENS. This problem can be solved with a simple retransmission approach: the publisher periodically re-initiates the procedure for building the timestamp until it receives a correct timestamp for the event. The same solution can be applied for subscription/unsubscription timestamp requests as well. Finally, the internal state of TMs should be preserved despite possible process failures in order to avoid possible TNO violations. This can be obtained by adopting standard replication techniques [20], [21].

Ordering Features - A total ordering imposed on event notifications is important to provide distinct subscribers with the same view of the evolution of an application. However, this kind of ordering does not capture the natural *cause-effect* relationship that may relate events. This kind of relationship may be extremely important in applications where subscribers are expected to observe a coherent evolution of events with respect to a given application-level semantics. Specifically, here we are interested in analyzing if our solution may be used to also guarantee the causal ordering of events produced by publishers [22]. This order is particularly relevant because it allows to recognize *cause-effect* relationships among published events [23].

Our solution is clearly able to guarantee causally ordered notifications within each single topic. If a publisher publishes two events in the same topic, in fact, it will sequentially request two timestamps for the same topic whose content, by construction, will guarantee the FIFO-order as defined at the publisher side. If a process is notified about an event and then publishes a new event in the same topic, this event's timestamp will be greater than the one attached to the notified event.

If events are published in different topics, their timestamps are generally not comparable, thus causal ordering cannot be enforced. Ensuring this ordering imposes some slight modifications to the ordering algorithm introduced in Section 3.4. In particular, the definition of $\mathcal{SG}_{\mathcal{T}}$ should be revised as follows: the *sequencing group* of a topic T ($\mathcal{SG}_T$) is a set of topics including T and all T' such that there is at least a subscription including both T and T'. This update changes the behaviour of the multistage sequencer and produces timestamps that, given an event e published on topic T, totally order it with respect to all events published in topics subscribed by subscribers that will be notified about e. Thanks to this change, we can guarantee causal order also in the case there is a subscriber subscribed to T and T', and a second subscriber of T that is notified of an event e in that topic, and afterwards publishes an event e' in topic T'. The new structure of the timestamps in this case guarantees that $\mathcal{SG}_{\mathcal{T}} \cap \mathcal{SG}_{\mathcal{T}'} \supseteq \{\mathcal{T}, \mathcal{T}'\}$, and thus that event e' published in T' will get a timestamp larger than the one associated to e.

5 Performance Evaluation

In this section we evaluate the performance of the proposed solution. First, we describe the system deployment, the metrics of interest for the evaluation, and the load used to stress the system. Then, we analyze how the characteristics of our timestamps (i.e., their sizes) vary depending on different application loads. Finally, we evaluate the overall performance of a prototype implementing our solution.

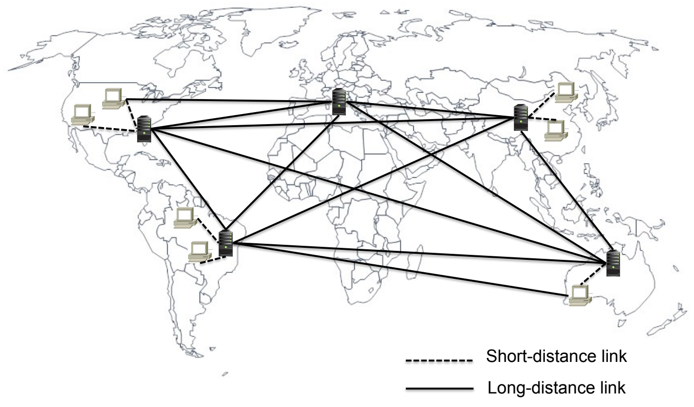

Fig. 6. A geo-distributed system interconnected by long-distance links. Clients connect to servers by means of two kinds of link: short- and long-distance.

5.1 System deployment

We consider the scenario depicted in Figure 6: a geo-distributed infrastructure using an asynchronous messaging layer to convey client updates that have to be applied in a consistent order. This order is provided by our ordering solution. Servers are located in dispersed geographic *sites* and connected to each other through high-latency links, while clients connect to servers through either low- or high-latency links. These links represent short- or long-distance WAN connections, respectively, and refer to the geographic distance between a client and a server or between two servers. We assume that the geo-distributed application running on top of this infrastructure implements an event-based communication pattern leveraging an underlying reliable topic-based publish/subscribe middleware. Our ordering solution is deployed on top of this middleware such that each server in the ENS hosts the TMs of the topics that are more popular in terms of publications and subscriptions in that geographic location. This represents a typical scenario, in which the topic popularity follows a locality principle, as the one shown in [13]. As an example, consider a geo-distributed system that disseminate news, results, and statistics about soccer. Servers in Italy, US, Brazil host the TMs of topics that are more popular in these countries, such as *teams, results, players, records, ...* of the Italian, American, and Brazilian leagues, respectively. A client in one of those countries issues with higher probability publications/subscriptions to topics related to the local league. In the next subsection we will discuss the publication/subscription model that captures this scenario.

We implement a prototype of the ordering protocol and deploy it on 5 physical servers. Each server is a Dell PowerEdge R210 II, with an Intel Xeon E3 1270v2 3.50 GHz processor and 16GB of memory. We consider a publish/subscribe system with 1000 topics, with the precedence relation $\rightarrow$ defined by the topic identifier. TMs are equally partitioned onto servers (i.e., each server hosts 200 TMs). We assume that TMs of topics with id 1-200 are on server 1, TMs of topics with id 201-400 are on server 2, and so on. We deploy 1000 clients into 5 physical machines. We use the *netem* network emulator to emulate short- and long-distance WAN links. A short-distance link has average delay of 10 ms and standard deviation of 2 ms, while a long-distance link has average delay of 100 ms and standard deviation of 10 ms.

5.2 Settings and metrics

The following metrics have been considered for our evaluation:

Timestamp size Number of entries in a timestamp.

End-to-end latency Time taken for the construction of a timestamp. This time includes the timestamp request by a publisher, the actual timestamp generation by TMs, and the response issued by some TM.

Throughput The number of timestamps generated by TMs in a time unit.

Percentage of outgoing bandwidth The fraction of bandwidth that a physical server that hosts one or more TMs reserves to forward timestamps to other servers during their generations. This fraction is computed as the number of bytes of timestamps forwarded to other servers divided the total number of bytes of all timestamps handled in that server during an experiment.

The tests were performed by varying two basic parameters: event rate (number of timestamp requests per time unit) and total number of subscribed topics (global sum of the size of all subscriptions).

The publication and subscription models *at each site* of the geo-distributed system were varied as follows:

Publication model We model publications as a probability distribution over the set of topics. We consider a *power-law* distribution with shape 0.901 or 0.349. The former refers to the 0.5% of topics having a probability of 80% to be selected for a new publication. The latter, instead, refers to the 40% of topics having a probability of 80% to be selected for a new publication;

Subscription model As for publications, subscriptions are modeled as a probability distribution over the set of topics. Again, we consider *power-law* distributions with shapes 0.901 and 0.349.

The distribution with shape 0.901 represents a model in which a few topic per each site have high popularity within a geographic region, and are subscribed with very high probability by clients of the same region. Hence, just a small percentage of the clients of a geographic region subscribe to topics that are outside their region. In other words, most of the content generated in a certain region is mostly consumed in the same region. This captures the *geographic topic popularity* pattern described in [13] in the context of YouTube video popularity. Considering the prototype deployment, this means that most of the computation for timestamp generation is local to each server. This property is confirmed in the performance study of Section 5.4.

The distribution with shape 0.349 represents a model in which a few topic per each site have high popularity within a geographic region. However, differently from the previous distribution, these topics are subscribed by clients that can be of any region as topics might diffuse to other geographical regions along time. This phenomenon captures the *spray-and-diffuse* pattern described in [13] and [14] in the context of YouTube videos and twitter hastags respectively. In this case, considering the prototype deployment, several geo-distributed servers could be involved in timestamp generation.

5.3 Timestamp size evaluation

We first analyze the scalability of the proposed timestamping technique over the number of subscribed topics by calculating the average size of timestamps. Note that this size only depends on issued subscriptions and is completely independent from the specific deployment scenario. For this reason, and only for the evaluation of this aspect, we simplified the setting described in section 5.1 by considering a single server hosting all the TMs. The evaluation is conducted in a system with 10000 clients and 1000 available topics, with the total number of subscribed topics that varies in the range [10k - 2000k]. Topic ranking was defined to match the topic popularity as defined by the publication and subscription models. Note that in this setting, geographic topic popularity and spray-and-diffuse boil down to popularity topic distribution with two different shapes.

Figure 7 shows the mean timestamp size by varying the number of subscribed topics. The leftmost point of both curves represents the case where each of the 10000 available clients, acting as subscribers, subscribe 10 topics. In both cases the average timestamp size is in the same order of magnitude of the number of subscribed

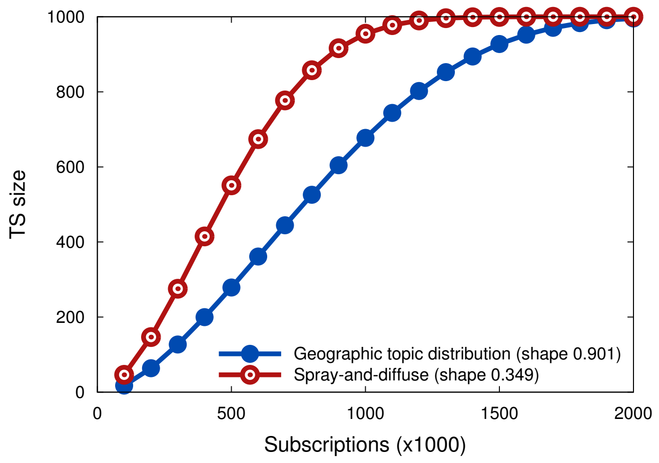

Fig. 7. Mean timestamp size varying the number of subscribed topics. This experiment considers all TM_i hosted in a single site.

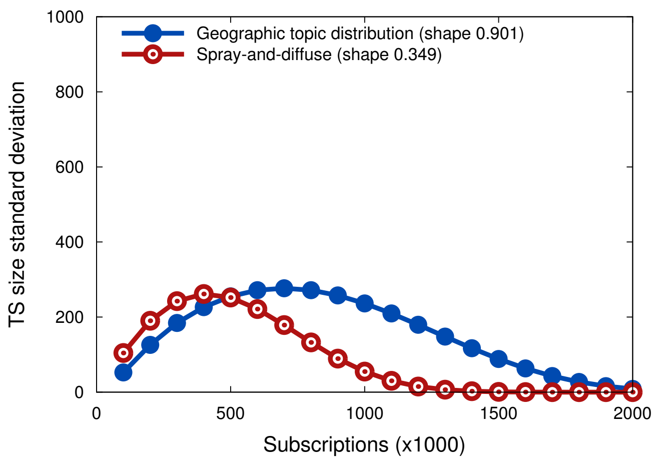

Fig. 8. Standard deviation of the mean timestamp size varying the number of subscribed topics. This experiment considers all TM_i hosted in a single site.

topics per clients (46 and 17 topics for the curves with shape 0.349 and 0.901 respectively). These size would both decrease to 0 by further reducing the amount of subscribed topics to 1 for subscriber as there would be no intersections among subscriptions. On the right side of the picture, as the number of topics subscribed grows approaching the maximum of 1000 per subscriber, both curves asymptotically approach their maximum. It is worth noticing that (i) more skewed popularity distributions (i.e., shape 0.901) produce smaller timestamps as less popular topics are rarely subscribed by more than one subscriber, and (ii) in both cases the size of the timestamps remains fairly small in meaningful scenarios where the number of topics subscribed per subscriber is not huge[3]. Fig 8 reports the standard deviation for the same experiments.

5.4 Prototype performance evaluation

Figure 9 shows the mean end-to-end latency by varying the event rate, i.e., the number of timestamp requests per second. The subscription size for each client is 10 topics (10k topics subscribed in total). Figure 9 evidences that our protocol imposes a small delay for timestamp generation when subscriptions include with very high probability topics whose TMs reside on the same physical host (power-law distribution with shape 0.901). In this case most of the computation for a timestamp generation is local as discussed in Section 5.2. In addition, according to the publication model, the timestamp requests are more likely to come from publishers that are geographically closer. Hence, low-distance links are used most of the time. On the contrary, when publication and subscription models follow the *spray-and-diffuse* pattern (power-law distribution with shape 0.349), more physical machines are involved in the timestamp generation and the mean end-to-end latency is affected by communication on long-distance links.

3. Typically, scenarios where most subscriber subscribe a vast majority of the available topics are better served by broadcast primitives rather than publish/subscribe ones.

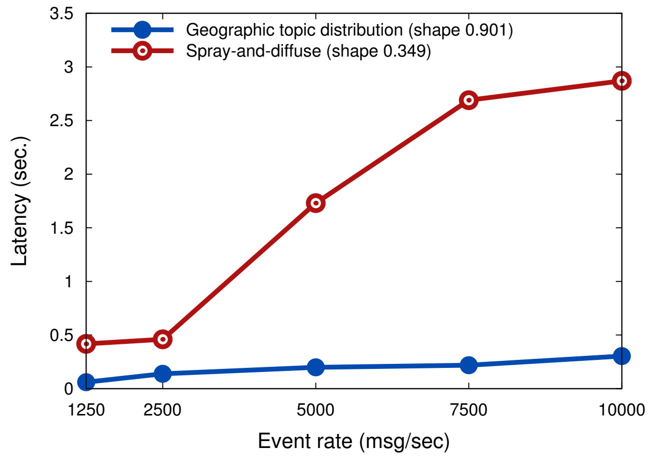

Fig. 9. Mean end-to-end latency for different publication and subscription models by varying the event rate.

Similar results are obtained when measuring the throughput, i.e., the number of timestamps generated in a second. Our protocol sports its peak performance in the proposed deployment when fed with 5000 timestamp requests per second for both probability distributions. When these requests follow the *geographic topic popularity* pattern the system shows its best performance, reaching a peak of more that 4k timestamps generated per second.

The rationale behind the results showed above is confirmed by Figures 11 and 12, which depict the percentage of outgoing bandwidth for each physical server in the presence of *geographic topic popularity* and *spray-and-diffuse* patterns, respectively. When submissions show *geographic topic popularity*, the TMs involved in the generation of a timestamp are likely to be hosted by the same physical server. Similarly, under that pattern, the vast majority of timestamp requests are issued on low-

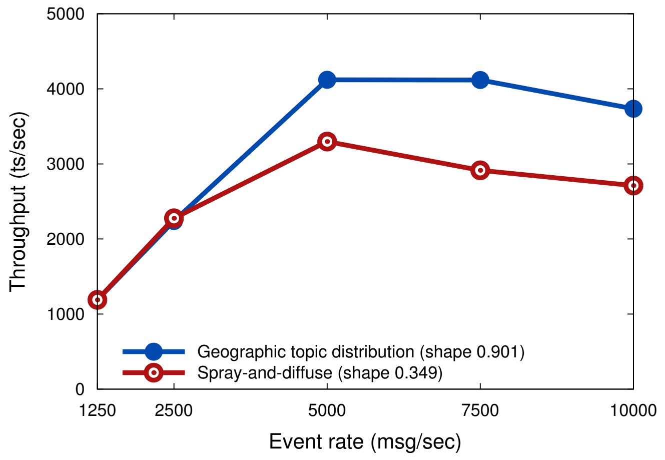

Fig. 10. Mean throughput for different publication and subscription models by varying the event rate.

distance WAN links. Figure 11 evidences how only TMs on servers 4 and 5 forward timestamp requests to TMs on different servers. Indeed, TMs on servers 4 and 5 are the TMs of topics with lower precedence according to their identifiers. The percentage of outgoing bandwidth is negligible for server 3 and null for servers 1 and 2.

On the contrary, when subscriptions and publications follow the *spray-and-diffuse* pattern, the percentage of outgoing bandwidth for each server is considerably higher (except for server 1).

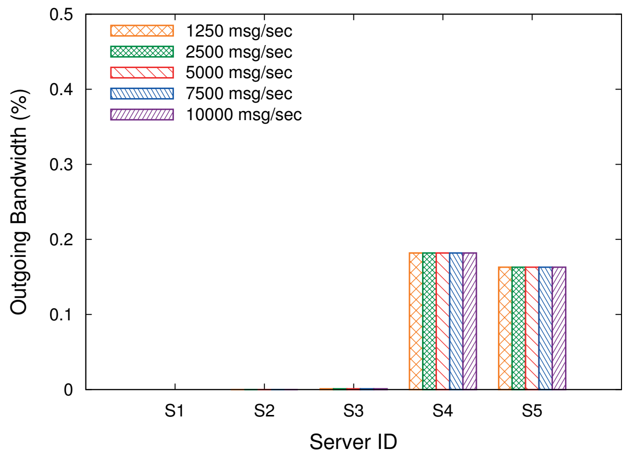

Fig. 11. Percentage of outgoing bandwidth in case of *geographic topic popularity* pattern (power-law distribution with shape 0.901).

Figures 13 and 14 show the mean end-to-end latency and throughput by varying the total number of subscriptions in the system and fixing the event rate to 5000 timestamp requests per second. As the previous experiments, even in this case the publication and subscription models affect the performance of the system. The interesting fact, however, is that the number of subscriptions does not impact the performance of the protocol when the subscription model follows the *geographic topic popularity* pattern. A larger number of subscriptions results just in some additional local computations. On the contrary, when subscriptions follow the *spray-and-diffuse* pattern, a larger number of subscriptions impose computations needed to generate timestamps which can involve multiple sites.

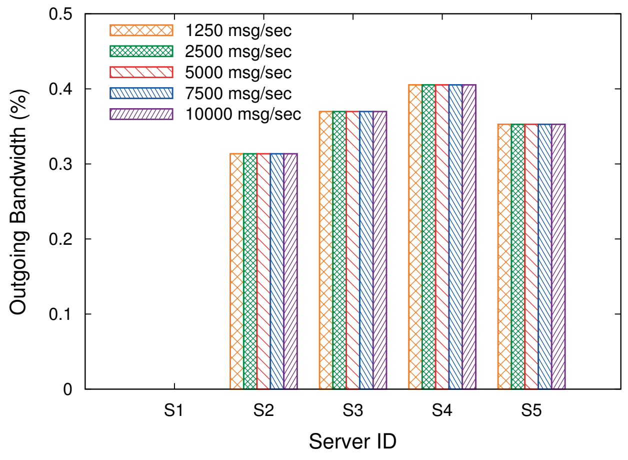

Fig. 12. Percentage of outgoing bandwidth in case of *spray-and-diffuse* pattern (power-law distribution with shape 0.349).

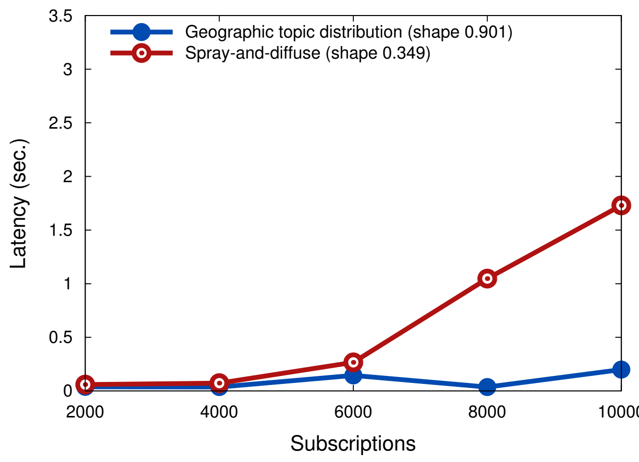

Fig. 13. Mean end-to-end latency for different publication and subscription models by varying the number of subscriptions.

6 Related work

Although many real world applications require support for QoS such as ordering, the majority of current publish/subscribe solutions mainly operates on a best-effort basis [24]. Among the QoS-enabled event dissemination services, JEDI [25] is a publish/subscribe middleware that satisfies a causal order delivery of events. It is obtained by means of a *return value*, i.e., a response

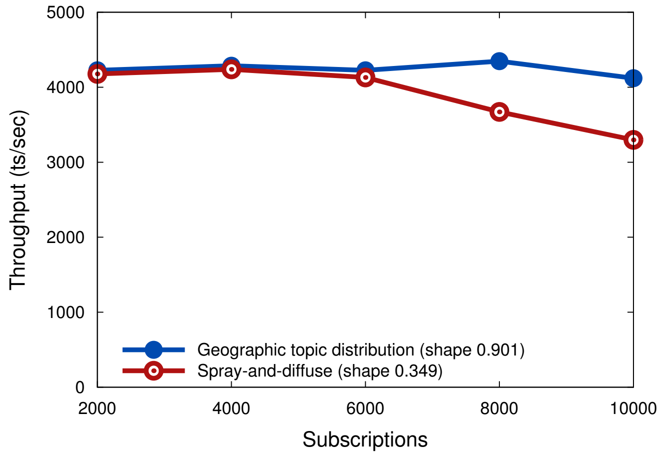

Fig. 14. Mean throughput for different publication and subscription models by varying the number of subscriptions.

message that a receiver uses to notify an event delivery to the producer. This mechanism is clearly not scalable in the presence of a high number of nodes and high event rate. A total order algorithms for publish/subscribe systems is presented in [2]. Similar to our work, authors use a *sequencing network* to order events across multiple groups of subscribers. However, their solution is not able to handle subscription dynamics. A new subscription/unsubscription can create loops in the sequencing network (*circular dependency problem*), which, thus, must be rebuilt from scratch. On the contrary, our solution solves these problems by defining a total order relation among topics that determines a one-way sequence of topic managers that establish an order for events.

Two interesting solutions for ordering in content-based publish/subscribe middleware recently appeared in [5] and [26]. The paper in [26] presents a partition-tolerant content-based publish/subscribe algorithm that can tolerate concurrent failures of brokers or communication links up to a value δ. However, this solution only guarantees a per-source FIFO order. The authors of [5] introduce a *Pairwise Total Order* specification that, in its *weak* variant, fully matches our TNO property. Both solutions follow the same path: discard liveness (delivery reliability) in favor of safety (ordering). However, the system presented in [5] supports the content-based event selection model by deeply integrating within the PADRES publish/subscribe system. Its applicability to other ENSs is thus limited. Conversely, our solution is completely decoupled from the underlying messaging layer and it can be easily adapted to several different contexts (i.e., ENS or group toolkit). In [5], the correct order of events is reconstructed by brokers, by using advertisements and subscriptions to detect *conflicts* and buffers to maintain them during the resolution phase. Conflicts are resolved through an acknowledgment mechanism. Note that this solution increases the load on brokers due to the widespread usage of ACK messages that enforce a tight synchronization among multiple brokers, hampering the ability of the system to support strong loads [11]. To this respect, our solution has been natively designed by enforcing a *one-way message flow* design within the multistage sequencer that, by removing explicit synchronization among TMs, helps supporting growing loads and thus scales gracefully even if the order of events must be reconstructed on subscribers' side. Recently, Yahoo started adopting HedWig [27] as a topic-based publish/subscribe solution for event dissemination within the PNUTS system. HedWig provides strong data delivery guarantees but, contrarily to our solution, it only enforces per-topic ordering within a single datacenter. Therefore, topics are independent in HedWig, as there is no way to order notifications related to different topics.

7 Conclusions

Totally ordering notifications produced by a publish/subscribe system deployed on top of a distributed infrastructure is a complex problem due to the inherent dynamism of pub/sub interactions, and to the fact that the order has to be achieved in a fully decoupled way, without any explicit synchronization among components of the systems itself.

This paper presented an ordering solution that can be used to order notifications delivered by a reliable event notification service to satisfy a *total notification order* specification. The main contribution is a timestamping mechanism that considers topics and overlapping subscriptions to automatically build appropriately sized timestamps that are attached to published events. Subscribers can use timestamps to infer the correct sequence of notifications to be enforced locally. The ordering mechanism uses a multistage sequencer to build timestamps. A precedence relationship among topics is used to determine the order in which stages are executed, i.e., to determine a one-way sequencing path (no loops or feedbacks that may create instability in the network) for the timestamp generation. Thanks to its internal structure the sequencer can be deployed in multiple flexible ways: several stages can be co-located in the same server of a distributed system thus making local to that server the entire generation of several timestamps.

The performance of the proposed solution has been evaluated through a prototype implementation. In particular, the results show that our solution is able to deliver very good performance for geo-distributed applications characterized by a load following a *geographic topic distribution* pattern where user interests are geographically clustered.

Acknowledgments

This work has been partially supported by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research, and by the academic project n. C26A133HZY funded by the University of Rome ”La Sapienza”. The authors would like to thank the anonymous reviewers for all their comments that helped to improve both presentation and content of the paper.

REFERENCES

[1] C. Liebig, M. Cilia, and A. P. Buchmann, "Event composition in time-dependent distributed systems," in *Proceedings of the 4th International Conference on Cooperative Information Systems (IFCIS), Edinburgh, Scotland, September 2-4*. IEEE, 1999, pp. 70–78.

[2] C. Lumezanu, N. Spring, and B. Bhattacharjee, "Decentralized message ordering for publish/subscribe systems," in *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (Middleware), Melbourne, Australia, November 27 - December 1*, 2006, pp. 162–179.

[3] A. Malekpour, A. Carzaniga, G. T. Carughi, and F. Pedone, "Probabilistic FIFO ordering in publish/subscribe networks," Faculty of Informatics, University of Lugano, Technical Report USI-INF-TR-2011-2, Tech. Rep., 2011.

[4] H. Garcia-Molina and A. Spauster, "Ordered and reliable multicast communication," *ACM Transaction on Computer Systems*, vol. 9, no. 3, pp. 242–271, 1991.

[5] K. Zhang, V. Muthusamy, and H.-A. Jacobsen, "Total order in content-based publish/subscribe systems," in *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS), Macau, China, June 18-21*. IEEE, 2012.

[6] P. R. Pietzuch, B. Shand, and J. Bacon, "Composite event detection as a generic middleware extension," *IEEE Network*, vol. 18, no. 1, pp. 44–55, 2004.

[7] A. R. Bharambe, S. G. Rao, and S. Seshan, "Mercury: a scalable publish-subscribe system for internet games," in *Proceedings of the 1st Workshop on Network and System Support for Games (NETGAMES), Braunschweig, Germany, April 16-17*. ACM, 2002, pp. 3–9.

[8] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky, "Hierarchical clustering of message flows in a multicast data dissemination system." in *IASTED PDCS*, 2005, pp. 320–326.

[9] RTI, "Real time messaging and integration middleware," http://www.rti.com/.

[10] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[11] K. Birman, G. Chockler, and R. van Renesse, "Toward a cloud computing research agenda," *SIGACT News*, vol. 40, no. 2, pp. 68–80, 2009.

[12] R. Barrett, "Transactions across datacenters," Google I/O developer conference http://www.google.com/events/io/2009/sessions/TransactionsAcrossDatacenters.html, 2009.

[13] A. Brodersen, S. Scellato, and M. Wattenhofer, "YouTube around the world: geographic popularity of videos," in *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012, pp. 241–250.

[14] K. Y. Kamath, J. Caverlee, K. Lee, and Z. Cheng, "Spatio-temporal dynamics of online memes: a study of geo-tagged tweets," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 667–678.

[15] R. Baldoni, S. Cimmino, and C. Marchetti, "A classification of total order specifications and its application to fixed sequencer-based implementations," *Journal of Parallel and Distributed Computing*, vol. 66, no. 1, pp. 108–127, 2006.

[16] X. Defago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computer Survey*, vol. 36, no. 4, pp. 372–421, 2004.

[17] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg, "Content-based publish-subscribe over structured overlay networks," in *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS), Columbus, OH, USA, June 6-10*. IEEE, 2005, pp. 437–446.

[18] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[19] K. Birman, "Rethinking multicast for massive-scale platforms," in *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS), Montreal, Canada, June 22-26*. IEEE Computer Society, 2009.

[20] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed systems (2nd edition)," S. Mullender, Ed. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, ch. The primary-backup approach, pp. 199–216.

[21] F. B. Schneider, "Distributed systems (2nd edition)," S. Mullender, Ed. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, ch. Replication management using the state-machine approach, pp. 169–198.

[22] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transaction on Computer Systems*, vol. 5, no. 1, pp. 47–76, 1987.

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commununication of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[24] G. Muehl, L. Fiege, and P. R. Pietzuch, *Distributed event-based systems*. Springer, 2006.

[25] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Transaction on Software Engineering*, vol. 27, no. 9, pp. 827–850, 2001.

[26] R. S. Kazemzadeh and H.-A. Jacobsen, "Partition-tolerant distributed publish/subscribe systems," in *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS), Madrid, Spain, October 4-7*. IEEE, 2011, pp. 101–110.

[27] HedWig, http://wiki.apache.org/hadoop/HedWig.

APPENDIX
ALGORITHM PSEUDO-CODE

This section illustrates the pseudocode of the algorithm. Before proceeding with the description, we introduce the local data structures maintained by publishers, subscribers, and topic managers.

Local data structures at each publisher p_i: each publisher maintains locally the following data structures:

- id_e: is a unique identifier associated to each event produced by p_i.
- $outgoingEvents_i$: a set variable, initially empty, storing the events indexed by event id that are published by the upper application layer, and that are waiting for being published in the ENS.

Local data structures at each subscriber s_i: each subscriber maintains locally the following data structures:

- $subs_i$: a set variable storing topics subscribed by p_i;
- sub_LC_i: a set of $< T_i, sn_i >$ pairs, where T_i is a topic identifier and sn_i is an integer value; sub_LC_i contains a pair for each topic $T_i \in subs_i$. Initially, for each topic $T_i \in subs_i$ the corresponding sequence number is $\perp$.
- $to\_deliver_i$: a set variable storing $< e, ts, T >$ triple where e is an event (not in right order) notified by the ENS, ts is the timestamp attached to the event and T is the topic where the event has been published.

Local data structures at each topic manager TM_{T_i}: Each topic manager maintains locally the following data structures:

- LC_{T_i}: is an integer value representing the sequence number associated to topic T_i, initially 0.
- LLC_{T_i}: is a set of $< T_j, sn_j >$ pairs where T_j is a topic identifier and sn_j is a sequence number. Such set contains an entry for each topic $T_j \in \mathcal{SG}_{T_i}$ such that $T_i \rightarrow T_j$.
- $externalSubs_{T_i}$: a set of $< id, sub >$ pairs where sub is a subscription (i.e., a set of topics $\{T_j, T_k \dots T_h\}$) and id is the subscriber identifier. Such a set contains all the subscriptions that include T_i.
- SG_{T_i}: is a set containing identifiers of topics belonging to the sequencing group of T_i.

As an example, let us consider a system where $S_i = \{T1, T2, T3\}$ and $S_j = \{T1, T2\}$ are respectively the two subscriptions of subscribers s_i and s_j. The three variables $externalSubs$ maintained by each topic manager are respectively: $externalSubs_{T1} = \{< i, S_i >, < j, S_j >\}$, $externalSubs_{T2} = \{< i, S_i >, < j, S_j >\}$, and $externalSubs_{T3} = \{< i, S_i >\}$.

The PUBLISH() **Operation.** The algorithm for a PUBLISH() operation is shown in Figure 16. To simplify the pseudo-code of the algorithm, we defined the following basic functions:

- generateUniqueEventID(e): generates a locally unique identifier for a specific event e.
- next(ts, T): given a timestamp ts and a topic identifier T, the function returns the identifier of the topic T' preceding T in the timestamp ts according to the precedence relation $\rightarrow$. If such topic does not exist, then the function returns $null$. If a $null$ value is passed as topic identifier, the function returns the last topic identifier contained in the timestamp.
- getTMAddress(T): returns the network address of the topic manager TM_T responsible for topic T.
- update($ts, < T, LC_T >$): updates the event timestamp ts changing the pair $< T, - >$ with the pair $< T, LC_T >$.
- updateLLC($LLC, < T, sn' >$): modifies the set LLC by updating the pair corresponding to topic T with a pair $< T, sn'' >$, where sn'' is the maximum between sn (the sequence number already stored locally in LLC for topic T) and sn' (i.e., the sequence number contained in the partially filled timestamp).

In addition, we have defined a more complex function, namely updateSequencingGroup($T, externalSubs_T$), that generates the sequencing group $\mathcal{SG}_T$ by considering the set of subscriptions containing T (i.e., subscriptions stored in $externalSubs_T$). The pseudo-code of the function is shown in Figure 15.

```
function updateSequencingGroup($T_i, externalSubs$):

(01)  for each $< -, s > \in externalSubs$ do $SG_{T_i} \leftarrow SG_{T_i} \cup s$; endfor
(02)  for each $T_j \neq T_i \in SG_{T_i}$ do
(03)        let $S = \{s \in externalSubs | T_j \in s\}$;
(04)        if ($|S| \leq 1$) then $SG_{T_i} \leftarrow SG_{T_i} / \{T_j\}$ endif
(05)  endfor
(06)  return $SG_T$.
```

Fig. 15. The updateSequencingGroup() function (for a topic manager TM_{T_i}).

When an event e is published in a topic T, the publisher p_i executes the algorithm shown in Figure 16(a). In particular, it associates with e a unique identifier generated locally (line 01), it puts the event, together with the topic and the corresponding identifier in a buffer (line 02) and sends a CREATE_PUB_TS (id_e, i, T) message to the topic manager TM_T, associated with T (lines 03-04).

Receiving the CREATE_PUB_TS (id_e, i, T) message, the topic manager TM_T executes the algorithm shown in Figure 16(b). In particular, it first creates an empty timestamp ts_e containing an entry $< T_j, \perp >$ for each topic T_j belonging to the sequencing group of T (line 02), it updates the sequence numbers of topics T_k following T in the topic ranking with the values locally stored in LLC_T (line 04), it increments its local sequence number and updates the corresponding entry in ts_e (lines 06 - 07). Finally, TM_T sends a FILL_IN_PUB_TS message containing the timestamp to the preceding topic

```
operation PUBLISH($e, T$):

(01)  $id_e \leftarrow$ generateUniqueEventID ($e$);
(02)  $outgoingEvents \leftarrow outgoingEvents \cup \{(e, id_e, T)\}$;
(03)  $dest \leftarrow$ getTMAddress($T$);
(04)  send CREATE_PUB_TS ($id_e, i, T$) to $dest$;
-----------------------------------------------------------------
(05)  when EVENT_TS ($ts, e_{id}$) is delivered:
(06)      let $< e, id_e, T >\in outgoingEvents$
(07)      such that ($e_{id} = id_e$);
(08)      ENSpublish($< e, ts >, T$);
(09)      $outgoingEvents \leftarrow outgoingEvents / \{< e, id_e, T >\}$.
```

(a) Publisher Protocol (for a publisher process p_i)

```
(01)  when CREATE_PUB_TS ($e_{id}, j, T$) is delivered:
(02)      for each $T_j \in SG_{T_i}$ do $ev\_ts \leftarrow ev\_ts \cup \{< T_j, \perp >\}$; endfor
(03)      for each $< T_j, sn >\in LLC_{T_i}$ do
(04)                $ev\_ts \leftarrow$ update($ev\_ts, < T_j, sn >$);
(05)      endFor;
(06)      $LC_{T_i} \leftarrow LC_{T_i} + 1$;
(07)      $ev\_ts \leftarrow$ update($ev\_ts, < T_i, LC_{T_i} >$);
(08)      $t' \leftarrow$ next($ts, T_i$);
(09)      if ($t' = null$)
(10)         then send EVENT_TS ($ev\_ts, e_{id}$) to $p_j$;
(11)              else $dest \leftarrow$ getTMAddress($t'$);
(12)              send FILL_IN_PUB_TS (ev_ts, $e_{id}$, j) to $dest$;
(13)      endif
-----------------------------------------------------------------
(14)  when FILL_IN_PUB_TS ($ts, e_{id}, j$) is delivered:
(15)      for each $< T_j, sn_j >\in ts$ such that $T_i \rightarrow T_j$ do
(16)         $LLC_{T_i} \leftarrow$ updateLLC($LLC_{T_i}, < T_j, sn_j >$);
(17)      endFor
(18)      $ts \leftarrow$ update($ts, < T_i, LC_{T_i} >$);
(19)      $t' \leftarrow$ next($ts, T_i$);
(20)      if ($t' = null$)
(21)         then send EVENT_TS ($ts, e_{id}$) to $p_j$;
(22)         else $dest \leftarrow$ getTMAddress($t'$);
(23)              send FILL_IN_PUB_TS ($ts, e_{id}, j$) to $dest$;
(24)      endif.
```

(b) TM Protocol (for a topic manager TM_{T_i} belonging to the multistage sequencer of T)

Fig. 16. The publish() protocol.

manager until ts_e has been completed and it is finally returned to the publisher. The publisher (Figure 16(a)) pulls the event from the buffer, attaches the timestamp and publishes both on the ENS (lines 06 - 09). Note that, when a topic manager receives a FILL_IN_PUB_TS message, it just attaches its local sequence number (line 18) and update its local LLC variable to keep track of the sequence numbers associated to topics with lower rank (lines 15 - 17).

The NOTIFY() **Operation.** When an event e is notified by the ENS, a subscriber s_i executes the algorithm shown in Figure 17. To simplify the pseudo-code of the algorithm, we defined the following basic functions:

- isNext(ts, LC): The function takes as parameter a timestamp ts and a local subscription clock LC and returns a boolean value. In particular, let k be the size of the timestamp, the function returns true if and only if there exist $k - 1 < T_i, sn_i >$ pairs in ts equal to those stored in LC and the last $< T_j, sn_j >$ pair in ts is such that $< T_j, sn' >\in LC$ and $sn_j = sn' + 1$.
- updateSubLC($sub\_LC, < T, sn' >$): modifies the set sub_LC by updating the pair corresponding to topic T with the pair $< T, sn'' >$ where sn'' is the maximum between sn (the sequence number already stored locally in sub_LC for topic T) and sn' (i.e., the sequence number contained in the timestamp).

```
upon ENSnotify($< e, ts >, T$):

(01)  if ($e$ = subscriptionUpdate)
(02)     then if ($\forall < T_j, v >\in sub\_LC_i \mid T_j \neq T,\ \exists < T_j, v + 1 >\in ts$)
(03)              then for each $< T_j, sn_j >\in ts$ such that ($T_j \in subs_i$) do
(04)                        updateSubLC($sub\_LC_i, < T_j, sn_j >$);
(05)                   endFor
(06)              else $to\_deliver_i \leftarrow to\_deliver_i \cup \{< e, ts, T >\}$;
(07)          endif
(08)  endif
(09)  if (($T \in subs_i$) $\wedge$ ($e \neq$ subscriptionUpdate))
(10)     then if (isNext($ts, sub\_LC_i$) = true)
(11)              then trigger notify ($e, T$);
(12)                   for each ($< T_j, v >\in sub\_LC_i$)
(13)                        if($\exists < T_j, v' >\in ts$)
(14)                          then updateSubLC($sub\_LC_i, < T_j, v >$);
(15)                        endif
(16)                   endfor
(17)              else $to\_deliver_i \leftarrow to\_deliver_i \cup \{< e, ts, T >\}$;
(18)          endif
(19)  endif
-----------------------------------------------------------------
(20)  when $\exists < e, ts, T >\in to\_deliver_i$|isNext($ts, sub\_LC_i$) = true
(21)     $to\_deliver_i \leftarrow to\_deliver_i \setminus \{< e, ts, T >\}$;
(22)     trigger notify ($e, T$)
(23)     for each ($< T_j, v >\in sub\_LC_i$)
(24)        if ($\exists < T_j, - >\in ts$)
(25)          then updateSubLC($sub\_LC_i, < T_j, v >$);
(26)     endFor
```

Fig. 17. The notify() protocol (for subscriber s_i).

A subscriber s_i first checks if the event e is a subscriptionUpdate event for some topic T (line 01). If so, the subscriber checks if all previous events published on its subscribed topics have been notified by comparing the subscription timestamp with the local subscription one. This is done by checking entry by entry that all the sequence numbers (except the one associated to T) stored in the local subscription timestamp are equal to the one contained in the event timestamp -1. In this case, s_i uses the event timestamp to update its local subscription clock sub_LC_i (lines 02 - 05), otherwise it buffers the event and processes it later.

If the event is a generic application event, s_i checks if the topic T of the notified event is actually subscribed and then checks if it has been notified by the ENS in the right order (line 10). If such condition is satisfied, the event e is notified to the application (line 11) and the local subscription clock sub_LC_i is updated with the sequence numbers contained in the event timestamp (lines 12 - 16).

On the contrary, if the event e is not in the right order, the subscriber s_i buffers e (line 17) and continuously checks, by comparing its timestamp with the local subscription clock, when it is in the right order (lines 20 - 26).

The SUBSCRIBE() **and** UNSUBSCRIBE() **Operations.** The algorithm for a SUBSCRIBE() operation is shown in Figure 18. To simplify the pseudo-code of the algorithm, in addition to the functions used in the PUBLISH() algorithm, we defined the createSubTimestamp(sub) function, that

```
operation SUBSCRIBE($T$):

(01)  ENSsubscribe($T$);
------------------------------------------------------------
(02) upon ENSsubscribeReturn($T$):
(03)  $subs_i \leftarrow subs_i \cup \{T\}$;
(04)  for each $T_j \in subs_i$ do $ts \leftarrow ts \cup \{T_j, \bot\}$ endFor
(05)  $t' \leftarrow$ next($ts, null$);
(06)  $dest \leftarrow$ getTMAddress($t'$);
(07)  send FILL_IN_SUB_TS ($ts, (subs_i \cup \{T\}), id$) to $dest$;
------------------------------------------------------------
(08)  when COMPLETED_SUB_VC ($ts, s$) is delivered:
(09)      $sub\_LC_i \leftarrow ts$;
(10)      $e \leftarrow$ subscriptionUpdate;
(11)      for each $T_j \in subs_i$ do ENSpublish($< e, ts >, T_j$) endFor;
(12)      trigger subscribeReturn($T$);
```

(a) Subscriber Protocol

```
(01) when FILL_IN_SUB_TS ($ts, sub, j$) is delivered:
(02)  $externalSubs_i \leftarrow$ update ($externalSubs_i, < j, sub >$);
(03)  $SG_{T_i} \leftarrow$ updateSequencingGroup($T_i, externalSubs_{T_i}$);
(04)  for each $< T_j, sn_j > \in ts$ such that $((T_i \rightarrow T_j) \wedge (T_j \in SG_{T_i}))$ do
(05)          $LLC_i \leftarrow LLC_i \cup \{< T_j, sn_j >\}$;
(06)  endFor
(07)  $LC_{T_i} \leftarrow LC_{T_i} + 1$
(08)  $ts \leftarrow$ update ($ts, < T_i, LC_{T_i} >$);
(09)  $t' \leftarrow$ next ($ts, T_i$);
(10)  if ($t' = null$)
(11)    then send COMPLETED_SUB_VC ($ts, s$) to $j$;
(12)    else $dest \leftarrow$ getTMAddress($t'$);
(13)         send FILL_IN_SUB_TS ($ts, s, j$) to $dest$;
(14)  endif
```

(b) TM Protocol

Fig. 18. The subscribe() protocol.

```
operation UNSUBSCRIBE($T, subID$):

(01)  $subs_i \leftarrow subs_i / \{T\}$;
(02)  for each $T_j \in subs_i$ do
(03)      $dest \leftarrow$ getTMAddress($T_j$);
(04)      send UPDATE_SUB ($subID, subs_i$) to $dest$;
(05)  endfor
(06)  $dest \leftarrow$ getTMAddress($T$);
(07)  send UPDATE_SUB ($subID, \emptyset$) to $dest$;
(08)  ENSunsubscribe($T$);
```

(a) Subscriber Protocol

```
when UPDATE_SUB ($id, s$) is delivered:
(01)  if ($s \neq \emptyset$)
(02)    then $externalSubs_i \leftarrow$ update($externalSubs_i, < id, s >$);
(03)    else $externalSubs_i \leftarrow externalSubs_i / \{< id, - >\}$;
(04)  endif;
(05)  $SG_{T_i} \leftarrow$ updateSequencingGroup($T_i, externalSubs_{T_i}$).
(06)  for each $< T_j, - > \in LLC_{T_i}$ such that ($T_j \notin SG_{T_i}$) do
(07)          $LLC_{T_i} \leftarrow LLC_{T_i} \setminus \{< T_j, - >\}$;
(08)  endFor.
```

(b) TM Protocol

Fig. 19. The unsubscribe() protocol.

creates an empty subscription timestamp, i.e., a set of $< T, sn >$ pairs, where T is a topic identifier and sn is the sequence number for T, initially set to $\bot$. The subscription timestamp contains a pair for each topic T of a subscription S.

When a subscriber s_i wants to subscribe a new topic T, it executes the algorithm shown in Figure 18(a). In particular, it first invokes the ENSsubsribe(T) operation to make the subscription active on the ENS (line 01) and blocks until such operation completes. When the subscription is active on the ENS, s_i adds the topic T to the list of subscribed topics and creates an empty subscription timestamp through the createSubTimestamp function (including also topic T). Then, it sends a FILL_IN_SUB_TS $(ts, (subs_i \cup \{T\}), id)$ message to fill the timestamp and to forward the new subscription to the topic manager TM_{T_k} responsible for the last topic in the subscription, according to the precedence relation $\rightarrow$ (lines 03-07).

Upon the delivery of a FILL_IN_SUB_TS message, each topic manager $TM_{T'}$ executes the algorithm shown in Figure 18(b). In particular, $TM_{T'}$ updates its $externalSubs_k$ variable with the new subscription (line 02), recomputes the sequencing group $\mathcal{SG}_{T'}$ (line 03) and updates the sequence numbers of topics with lower rank (lines 04 - 06), increments its local sequence number (line 07), updates its entry in the subscription timestamp (line 08), and finally forwards the FILL_IN_SUB_TS message to the preceding topic manager until it is completed and returned to the subscriber.

When the subscriber s_i receives the completed subscription timestamp, it updates its local subscription clock (line 09) and publishes a subscriptionUpdate event on each topic T_j belonging to its subscription to keep other subscribers aligned with the subscription local clock (lines 10 - 11). Finally, it returns from the operation to notify the application that the subscription is now active (line 12).

The algorithm for the UNSUBSCRIBE() operation is shown in Figure 19. A subscriber that wants to unsubscribe from a topic T, removes it from the set of subscribed topics (line 01) and, then, informs all topic managers of these topics with the updated subscription through an UPDATE_SUB message (lines 02-05), including the topic manager of T that will receive an empty subscription (lines 06-07). When receiving an UPDATE_SUB message (Figure 19(b)), topic managers update the $externalSubs$ set accordingly with the received subscription, recompute the sequencing group, and remove topics that are no more in the sequencing group from the LLC_T set.

APPENDIX
CORRECTNESS PROOF

In this section we will show that the TNO property holds for any pair of events.

Definition 5: Given a generic subscriber s_i, let us denote $\tau_n(i, e)$ the time instant in which s_i is notified of e by the system.

Lemma 1: Let e_1 and e_2 be two events both published in a topic T. If a subscriber s_i notifies e_1 before e_2 and the sequencing group $\mathcal{SG}_T$ does not change during the two publish operations, then any other subscriber that notifies both e_1 and e_2 will notify e_1 before e_2.

Proof Let us suppose by contradiction that there exist two subscribers, namely s_i and s_j, and that s_i notifies e_1 and then e_2 (i.e., $\tau_n(i, e_1) < \tau_n(i, e_2)$), while s_j notifies e_2 and then e_1 (i.e., $\tau_n(j, e_2) < \tau_n(j, e_1)$).

Given a generic subscriber s_x notifying both e_1 and e_2, it follows that at time $\tau_n(x, e_1)$, $T_1 \in S_x$ and at time $\tau_n(x, e_2)$, $T_2 \in S_x$. Moreover, $sub\_LC_x \leq ts_{e_1}$ at time $\tau_n(x, e_1)$ and $sub\_LC_x() \leq ts_{e_2}$ at time $\tau_n(x, e_2)$.

Given the two timestamps ts_{e_1} and ts_{e_2} associated respectively with e_1 and e_2, let us first consider how they have been created and then let us show that it is not possible to have inversions in the notification order.

Let p_k and p_h be respectively the publishers of events e_1 and e_2. When a publisher publishes an event, it executes line 04 of Figure 16(a) and it sends a CREATE_PUB_TS message. Let us assume, without loss of generality, that TM_T delivers first the CREATE_PUB_TS message sent by p_k and then the CREATE_PUB_TS message sent by p_h.

Let v be the value of LC_T at TM_T when it delivers the CREATE_PUB_TS message sent by p_k. When TM_T delivers such a message, it creates an empty event timestamp ts_{e_1}, adds to ts_{e_1} the pairs $< T_j, sn_j >$ stored locally in LLC_T and containing sequence numbers associated with all the topics T_j preceding T in $\mathcal{SG}_T$ according with the topic rank, increments its local clock (i.e., $LC_T = v + 1$), and includes the pair $< T, v + 1 >$ in ts_{e_1} (lines 03 - 04, Figure 16(b)). Two cases can happen:

1) ts_{e_1} **contains only the entry for** T **and for topics in** LLC_T **(line 05)**: $ts_{e_1} = \{< T, v+1 >, < T_j, x > \cdots < T_j, y >\}$ (with $< T_j, x > \cdots < T_j, y > \in LLC_T$) and TM_T returns the completed timestamp to the publisher for the publication in the ENS.
2) ts_{e_1} **contains more entries (lines 06 - 09)**: in this case, there exists a topic T' following T in the topic order and TM_T sends a FILL_IN_PUB_TS message to $TM_{T'}$. Receiving such a message, $TM_{T'}$ just updates the pair $< T', \perp >$ contained in ts_{e_1} with the current value of $LC_{T'}$ and checks if there exists a topic T'' following T' in the timestamp. If so, it forwards the FILL_IN_PUB_TS message to $TM_{T''}$, otherwise, it returns ts_{e_1} to the publisher (lines 11 - 16).

When TM_T delivers the CREATE_PUB_TS message sent by p_h, it follows the same steps: it creates a template ts_{e_2} for the timestamp, increments its local sequence number (i.e., $LC_T = v + 2$), includes the pair $< T, v + 2 >$ in ts_{e_2}, and sends the timestamp to the publisher or to the following topic manager.

Let us note that TM_T updates sequence numbers of topics stored in LLC_T by following a monotonic increasing order, i.e., it always takes the maximum between the one already stored locally and the one in the received timestamp (cfr. lines 15-17, Figure 16(b)).

Considering that the sequencing groups are not changing, the timestamp will always include the same entries, i.e., ts_{e_1} and ts_{e_2} contain a set of pairs differing only for the sequence numbers associated with each topic. In particular, considering that (i) a topic manager can only increment its local sequence number when a publication occurs, and (ii) topic managers are connected through FIFO channels, it follows that for each topic T_i the sequence number v' associated with T_i in ts_{e_2} cannot be smaller than the one associated with T_i in ts_e. Therefore, $ts_{e_1} < ts_{e_2}$.

Considering that (i) as soon as an event e is notified to the application layer, the local subscription clock of the subscriber is updated according to the event timestamp (line 14, Figure 17), and that (ii) $ts_{e_1} < ts_{e_2}$, we have that s_j evaluating the notification condition at line 09 will store the event e_2 in the $to\_deliver_j$ buffer. This leads to a contradiction as e_2 will never be notified before e_1.

$\square_{Lemma\ 1}$

Lemma 2: Let e_1 and e_2 be two events published respectively in topics T_1 and T_2, with $T_1 \neq T_2$. Let us assume that the sequencing groups $\mathcal{SG}_{T_1}$ and $\mathcal{SG}_{T_2}$ do not change during the two publish operations. If a subscriber s_i notifies e_1 before e_2 then any other subscriber that notifies both e_1 and e_2 will notify e_1 before e_2.

Proof For ease of presentation and without loss of generality, let us assume that T_1 and T_2 are the only two topics subscribed by both s_i and s_j[4] (i.e., $\{T_1, T_2\} \subseteq S_i, S_j$).

Let us suppose by contradiction that there exist two subscribers, namely s_i and s_j, that notify both e_1 (published in topic T_1) and e_2 (published in topic T_2) but s_i notifies e_1 and then e_2 (i.e., $\tau_n(i, e_1) < \tau_n(i, e_2)$), while s_j notifies e_2 and then e_1 (i.e., $\tau_n(j, e_2) < \tau_n(j, e_1)$).

Given a generic subscriber s_x, if it notifies both e_1 and e_2, it follows that, at time $\tau_n(x, e_1)$, $T_1 \in S_x$ and at time $\tau_n(x, e_2)$, $T_2 \in S_x$. Moreover, $sub\_LC_x \leq ts_{e_1}$ at time $\tau_n(x, e_1)$ and $sub\_LC_x \leq ts_{e_2}$ at time $\tau_n(x, e_2)$.

Given the timestamps ts_{e_1} and ts_{e_2} associated respectively with e_1 and e_2, let us first consider how they have been created and then let us show that it is not possible to have inversions in the notification order.

Without loss of generality, let us assume that T_1 has higher precedence than T_2 in the topic order.

Considering how sequencing groups (and consequently timestamps) are defined by the updateSequencingGroup function shown in Figure 15, each event published in T_1 will have attached a timestamp containing the pairs $< T_1, x_1 >, < T_2, x_2 >$ and each event published in T_2 will have attached a timestamp containing the pairs $< T_1, y_1 >, < T_2, y_2 >$.

4. The proof can be easily extended to multiple intersections, by iterating the reasoning for any pair of topics that appears in more than one subscription.

When e_2 is published by the application layer, the publisher sends a CREATE_PUB_TS request for the event timestamp to TM_{T_2} (line 04, Figure 16(a)). Receiving such a request, TM_{T_2} executes line 02 of Figure 16(b) and creates an empty event timestamp containing entries for T_1 and T_2 (i.e., $ts_{e_2} \supseteq < T_1, \perp >, < T_2, \perp >$), increments its local clock, let's say to a value v (line 06), updates its component of the timestamp with its local clock (i.e., $ts_{e_2} \supseteq < T_1, \perp >, < T_2, v >$), and sends a FILL_IN_PUB_TS message containing ts_{e_2} to the following topic manager selected in the event timestamp according to the precedence relation $\rightarrow$ (i.e., to TM_{T_1}). Delivering such message, TM_{T_1} will execute lines 15 - 17, Figure 16(b) by storing locally the pair $< T_2, v >$ in its LLC_1 variable.

The same procedure is executed when e_1 is published. Note that, since $T_1 \rightarrow T_2$, it follows that the pair $< T_2, v >$ contained in ts_{e_2} will be attached to ts_{e_1}.

In the worst case scenario, due to concurrency in the timestamp creation procedure, TM_{T_1} can either deliver first the CREATE_PUB_TS message sent from the publisher of e_2 and then the FILL_IN_PUB_TS message sent by TM_{T_2} or vice-versa, it can first manage the FILL_IN_PUB_TS message and then the CREATE_PUB_TS one:

1) **TM_{T_1} delivers the CREATE_PUB_TS message for event e_1 and then the FILL_IN_PUB_TS message for ts_{e_2}.** Delivering the CREATE_PUB_TS for event e_1, TM_{T_1} creates an empty event timestamp for e_1 (i.e., $ts_{e_1} \supseteq < T_1, \perp >, < T_2, \perp >$), updates the entry related to T_2 with the pair $< T_2, v' >$ stored locally in LLC_1 (with $v' \leq v$), updates its local clock to $v_1 + 1$, updates its timestamp component with its local clock (i.e., $ts_{e_1} \supseteq < T_1, v_1 + 1 >, < T_2, v' >$), and sends a FILL_IN_PUB_TS request containing ts_{e_1} to the following topic manager in the topic order (if any) or directly to the publisher. Delivering the FILL_IN_PUB_TS message for ts_{e_2}, TM_{T_1} executes line 11 of Figure 16(b), and updates its timestamp component with its local clock (i.e., $ts_{e_2} \supseteq < T_1, v_1 + 1 >, < T_2, v >$). Then, it sends a FILL_IN_PUB_TS request containing ts_{e_2} to the following topic manager in the topic order (if any) or directly to the publisher.
2) **TM_{T_1} delivers the FILL_IN_PUB_TS message for ts_{e_2} and then the CREATE_PUB_TS message for event e_1.** Delivering the FILL_IN_PUB_TS message for ts_{e_2}, TM_{T_1} executes line 18 of Figure 16(b), updates its timestamp component with its local clock (i.e., $ts_{e_2} \supseteq < T_1, v_1 >, < T_2, v >$), and updates its LLC_1 variable by storing the pair $< T_2, v >$. Then, it sends a FILL_IN_PUB_TS request containing ts_{e_2} to the following topic manager in the topic order. On the contrary, delivering the CREATE_PUB_TS for event e_1, TM_{T_1} creates the template for the event timestamp (i.e., $ts_{e_1} \supseteq < T_1, \perp >$), updates the entry related to T_2 with the pair $< T_2, v >$ stored locally in LLC_1, updates its local clock to $v_1 + 1$, updates its timestamp component with its local clock (i.e., $ts_{e_1} \supseteq < T_1, v_1 + 1 >$), and sends a FILL_IN_PUB_TS request containing ts_{e_1} to the following topic manager in the topic order (if any) or directly to the publisher.

Let us now consider the behavior of s_i and s_j when the notification is triggered by the ENS.

- **Subscriber s_i.** At time $\tau_n(i, e_1)$, s_i notifies e_1 and then updates its local clock by executing lines 04 - 11 of the notification procedure. In particular, s_i updates sub_LC_i with the pair $< T_1, v_1 >$ (or $< T_1, v_1 + 1 >$). At time $\tau_n(i, e_2)$, s_i is notified by the ENS about e_2. Since it has updated only the sub_LC_i entry corresponding to e_1, and considering that the value v has been assigned to T_2 for e_2, it means that $sub\_LC_i \leq ts_{e_2}$ and also e_2 can be notified.
- **Subscriber s_j.** At time $\tau_n(j, e_2)$, s_j receives e_2 that contains the entry $< T_1, v_1 + 1 >$ associated to e_1. However, since e_1 has not yet been received, the sub_LC_i data structure is not updated and the isNext() function returns false. As a consequence, s_j will buffer e_2 for a future notification when e_1 will be notified, and we have a contradiction.

$\square_{Lemma\ 2}$

Lemma 3: Let s_i be a subscriber that invokes a subscribe(T) operation at time t. If the ENS is reliable, then s_i eventually generates the subscribeReturn(T) event.

Proof The subscribeReturn(T) event is triggered by a subscriber s_i in line 12, Figure 18(a) when it delivers a COMPLETED_SUB_VC message. Such message is generated by the topic manger TM_{T_k} responsible of the highest ranked topic in the subscription of s_i (line 11, Figure 18(b)) when delivering a FILL_IN_SUB_TS message. Such message is originally generated by the subscriber itself and then forwarded by topic managers responsible of topics in the subscription (line 13, Figure 18(b)). Considering that (i) the FILL_IN_SUB_TS message is generated by the subscriber after the subscription is active on the ENS, (ii) the ENS guarantees that eventually such event happens, and (iii) messages are not lost in the forwarding chain, we have that the claim simply follows.

$\square_{Lemma\ 3}$

Lemma 4: Let s_i be a subscriber that invokes a subscribe(T) operation at time t and let $t + \Delta$ be the time at which the subscribe operation terminates. If the ENS is reliable, then s_i will notify all the events published in T at time $t' > t$.

Proof Let us suppose by contradiction that there exists an event e published in the new subscribed topic T after the subscription operation ends and that s_i never notifies such event. When the subscription terminates, the subscriber s_i updates its local subscription clock with the pairs contained in the subscription timestamp (line 09, Figure 18(a)). Let $sub\_LC_i = \{< T_i, v_i >, < T, v > \cdots < T_k, v_k >\}$ be such local subscription clock at time $t + \Delta$. Let us consider now the first event e published after time $t + \Delta$. When e is published, the publisher executes the algorithm in Figure 16 requesting topic managers in its sequencing group to fill in its timestamp. In particular, TM_T creates the timestamp and fills in its entry by incrementing its sequence number and adding the pair $< T, v + 1 >$. When such event is notified to s_i, it checks if the event can be notified immediately as it is the next with respect to the local subscription timestamp. Two cases can happen:

1) isNext(ts, sub_LC_i) = true. In this case, the event is immediately notified and we have a contradiction.
2) isNext(ts, sub_LC_i) = false. In this case, the event is stored in the $to\_deliver_i$ buffer while the subscriber waits until this events will be the next to be notified. Let un note that the ENS is reliable. Thus, published events will be eventually notified and the local subscription clock will be increased until the condition becomes true and the claim follows.

$\Box_{Lemma\ 4}$

Lemma 5: Let $\mathcal{SG}_{T_1}, \mathcal{SG}_{T_2}, \cdots .\mathcal{SG}_{T_n}$ be the sequencing groups of topics $T_1, T_2, \ldots T_n$ at time t. Let s_i be a subscriber that invokes subscribe(T) at time t and let $\mathcal{SG}'_{T_1}$, $\mathcal{SG}'_{T_2}$, $\ldots \mathcal{SG}'_{T_n}$ be the sequencing groups after the subscription operation ends at time $t + \Delta$ (i.e., when s_i triggers subcribeReturn(T)). For each T_i, $\mathcal{SG}_{T_i} \subseteq \mathcal{SG}'_{T_i}$.

Proof For each topic T_i, its sequencing group $\mathcal{SG}_{T_i}$ is calculated by considering the union of all the subscriptions containing T_i, stored locally at TM_{T_i} in the $externalSubs_i$ variable, and will include all the topics T_j such that there exist at least two subscriptions including both T_i and T_j (lines 02- 05, Figure 15).

At time t, when a subscriber s_i invokes a subscribe(T) operation, it executes the protocol in Figure 18(a) and will advertise that its subscription is changed by sending a FILL_IN_SUB_TS message that will flow through all the TMs responsible for topics in the subscription, from the last to the first in the topic order.

When a topic manager TM_j delivers such FILL_IN_SUB_TS message at a certain time $t' \in [t, t + \Delta]$, it will update its $externalSubs_j$ set with the new subscription (line 02, Figure 18(b)) and will use this set to compute the new sequencing group $\mathcal{SG}'_{T_j}$ when creating timestamps.

For each of these topics T_j, updating the $externalSubs_j$ variable with a subscription including T may cause a change of $\mathcal{SG}_{T_j}$ and the following cases can happen:

1) $\mathbf{T \in \mathcal{SG}_{T_j}}$ **at time t**. If $T \in \mathcal{SG}_{T_j}$, it means that $T \in externalSubs_j$ at time t. Thus, the new subscription of s_i has no effect on it, $\mathcal{SG}_{T_j} = \mathcal{SG}'_{T_j}$ and the claim follows.
2) $\mathbf{T \notin \mathcal{SG}_{T_j}}$ **at time t**. Two further cases can happen:
   a) $\mathbf{T \notin externalSubs_j}$ **at time t**. At time t, T was not included in $externalSubs_j$, meaning that there were no subscriptions containing together both T and T_j. As a consequence, the new subscription will be the only one including both topics. As a consequence, T will be not considered in the computation of the new sequencing group for T_j (lines 02-05, Figure 15), $\mathcal{SG}_{T_j} = \mathcal{SG}'_{T_j}$ and the claim follows.
   b) $\mathbf{T \in externalSubs_j}$ **at time t**. In this case, since $T \notin \mathcal{SG}_{T_j}$ at time t, it means that there exists at most another subscription including both T and T_j. As a consequence, the addition of the new subscription containing T to $externalSubs_j$ creates a pair of subscription sharing at least two same topics. The protocol will include T in the new sequencing group (lines 02- 05, Figure 15). Note that the insertion of a topic in $externalSubs$ can not remove any other topic already part of the sequencing group. As a consequence, in this case we will have that $\mathcal{SG}_{T_j} \subset \mathcal{SG}'_{T_j}$ and the claim follows.

$\Box_{Lemma\ 5}$

Lemma 6: Let s_i be a subscriber that invokes a subscribe(T) operation at time t and let $\mathcal{SG}'_T$ be the sequencing group of T after the subscription. If s_i notifies an event e published on the topic T, then the timestamp for e has been built according to $\mathcal{SG}'_T$.

Proof Subscriber s_i triggers the ENSsubscribe() only after it has inserted the topic T in its subscription (lines 07-08, Figure 18(a)) and has updated its local clock sub_LC_i. In particular, this happens when s_i delivers a COMPLETED_SUB_VC (ts, s) message (line 05, Figure 18(a)). The received subscription timestamp ts contains an entry for each T in the subscription of s_i. In particular, each entry is filled in with the current local clock LC_i incremented by one (line 03, Figure 18(b)) when TM_{T_i} delivers a FILL_IN_SUB_TS and updates its current sequencing group.

Let us show in the following that for any event e published on the topic T and notified to s_i, its timestamp will be generated following the new $\mathcal{SG}'_T$.

If e has been notified by s_i, then each entry of its local clock sub_LCi is smaller or equal than the corresponding entry in the timestamp ts_e of e, and it is strictly smaller for at least one entry. Considering that filling in ts_e each TM copies (or increments and copies) the entry in the timestamp, it follows that every TM has first filled in the subscription timestamp and then ts_e. Considering that (i) the subscription takes effect at each TM just after FILL_IN_SUB_TS message is delivered, and (ii) this message induces a change on the sequencing group, it follows that ts_e is created and filled in according to $\mathcal{SG}'_T$ and the claim follows.

$\square_{Lemma\ 6}$

Theorem 1: Let e_k and e_h be two events. If a subscriber s_i notifies first e_k and then e_h, any other subscriber that notifies both e_k and e_h will notify e_k before than e_h.

Proof The proof trivially follows from Lemmas 1 - 6 and considering that when a subscriber s_i invokes an unsubscribe(T) operation, TNO violations cannot happens because T is immediately removed from the subscription (line 01, Figure 19(a)) and the notification condition becomes false (line 01, Figure 17).

$\square_{Theorem\ 1}$